An Holistic Approach to Migrate Industrial Legacy Systems

Santiago Bragagnolo

▶ To cite this version:

Santiago Bragagnolo. An Holistic Approach to Migrate Industrial Legacy Systems. Programming Languages [cs.PL]. Université de Lille, 2023. English. NNT : 2023ULILB010 . tel-04337686v2

HAL Id: tel-04337686
https://theses.hal.science/tel-04337686v2

Submitted on 12 Dec 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Lille

Département de formation doctorale en informatique — École doctorale MADIS Lille
UFR IEEA

A Holistic Approach to Migrate Industrial Legacy Systems

Une approche holistique pour la migration des systèmes légataires industriels.

THÈSE

présentée et soutenue publiquement le 17 Mai 2023

pour l'obtention du

Doctorat de l'Université de Lille

(spécialité informatique)

par

Santiago Pablo Bragagnolo

Composition du jury

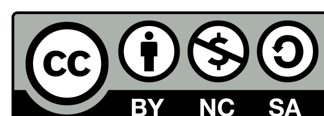
Président :	Alain Plantec	Professeur des universités – LAB-STICC
Rapporteurs :	Mireille Blay-Fornarino	Professeur des universités – Laboratoire I3S - CNRS - UNS
	Olivier Barais	Professeur des universités – IRISA Rennes
Examinateurs :	Gordana Rakić	Assistant professor – Faculty of Sciences, University of Novi Sad
	Alain Plantec	Professeur des universités – LAB-STICC
Directeurs de thèse :	Stéphane DUCASSE	Directeur de Recherche Univ. Lille, CNRS, Inria, CRIStAL
	Nicolas ANQUETIL	Maître de conférences Univ. Lille, CNRS, Inria, CRIStAL
Invités :	Abderrahmane SERIAI	Encadrant de these industriel – Berger-Levrault
	Christophe BORTOLASO	Encadrant de these industriel – Berger-Levrault

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
INRIA Lille - Nord Europe

Berger Levrault BL — cnrs — CRIStAL Centre de Recherche en Informatique, Signal et Automatique de Lille — Inria — 6 3 1 — RMod — Université de Lille

Copyright © 2023 by Santiago Pablo Bragagnolo

This work is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 3.0 Unported" license.



Acknowledgments

Each person crossing our lives is to be acknowledged in any enterprise where we use our knowledge, as we are our own experience: where we come from, what we discuss and do. However, there are some people without whom I can guarantee this would never happen.

I start by thanking you, dear reader. Without you, this would be a shout to the void, a mace without quarry (una maza sin cantera).

To my wife Angela, who supported me and walked this whole path with me, the person that saw me working days and nights, weeks and weekends, holy and unholy days.

To my directors, Nicolas Anquetil and Stephane Ducasse, who guided my path in strict science and still allowed a bit of my madness in this work.

To my loving parents, Luis and Renee, who gave me all the opportunities and bequeathed me the love for action, knowledge, intelligence and diligence.

To my blood siblings, Renee, Marcos and Lucia, who templated my character by fighting and making me angry, as the great siblings they are.

To my chosen siblings, Hernan de Gennaro, Gerardo Grimaldi, Eric Campastri and Angel Polutranka. Who knows where I would be without their humanity in those weird teenage years?

To my teachers and friends, Carlos Lombardi and Nicolas Passerini, who guided and defied my learning process. Beautiful people that dared to dream a dream that made us dream too.

To my brothers in arms, Guille Polito, Esteban Lorenzano, and Pablo Tesone, with whom we set sail together these many years on the same ship to god knows where.

My work is as mine as it is yours.

Abstract

Abstract

Context This project takes place in the context of a collaboration with Berger-Levrault, a large software company providing several services and applications developed with different technologies. In a business-critical quest for technological unification, cost reduction, and access to new markets, Berger-Levrault started an extensive program of software modernisation.

Problem Berger-Levrault has more than 90 applications written in Microsoft Access. We work on migrating an extensive Microsoft Access application, with nearly 20 years of development, to a web-technology-based solution. As Microsoft Access is a rich language, the project entails the migration of language, library, infrastructure, paradigm, user interface and architecture.

Such migration cannot be adequately split into multiple independent, successful migrations, as language migration directly affects all the other migrations. Furthermore, the solutions for each kind of migration vary in knowledge and technological requirements; no single homogenous approach could tackle this kind of migration, which pushes us to think of a higher-order solution.

Solution (contribution) This thesis presents (i) A reverse engineering approach can extract models from Microsoft Access. (ii) A migrating meta-model able to represent multiple languages. (iii) A set of migration metrics measuring the technological gap between the source system and the expected target. (iv) Visualisations to gain insight into the required work to make possible an architectural migration. (v) An interactive iterative approach to software migration allowing multiple destinations to migrate. (vi) We provide a rule-based migration engine allowing: partial migrations, based on the immediate and delayed application of contextualised rules, over multiple targets.

Validation and results To validate our solutions, we conduct multiple experiments yielding encouraging results. (i) We validate our reverse engineering over 10 projects by measuring information loss, proving that we can migrate with the available information. (ii) We validate our modelling approach over 34 projects by representing five technologies and still detect errors in a model of a specific language. (iii-iv) We use metrics and visualisations to help build source code and architectural blueprints and write migration feasibility reports for two large industrial projects. We also use the models and metrics to profile the used libraries to guide the construction of a testing application representing the usage of the libraries

based on the Pareto rule. (v-vi) We validated over a full form migration to back-end Java SpringBoot back end and Typescript Angular front-end. Migrating 47 tables and queries to Java and Typescript and 53 library artefacts to Java and Pharo.

Keywords: Software Migration, Software Modernisation, Language Transformation, Metrics, Visualisations, Industrial

Résumé

Contexte Ce projet se déroule dans le cadre d'une collaboration avec Berger-Levrault, une grande entreprise de logiciels fournissant plusieurs services et applications développés avec différentes technologies. Dans un souci d'unification technologique, de réduction des coûts et d'accès à de nouveaux marchés, Berger-Levrault a lancé un vaste programme de modernisation de ses logiciels.

Problème Berger-Levrault possède plus de 90 applications écrites en Microsoft Access. Nous travaillons sur la migration d'une application Microsoft Access étendue, avec près de 20 ans de développement, vers une solution basée sur la technologie web. Comme Microsoft Access est un langage riche, le projet implique la migration du langage, de la bibliothèque, de l'infrastructure, du paradigme, de l'interface utilisateur et de l'architecture.

Une telle migration ne peut être divisée en plusieurs migrations indépendantes et réussies, car la migration du langage affecte directement toutes les autres migrations. En outre, les solutions pour chaque type de migration varient en termes de connaissances et d'exigences technologiques ; aucune approche homogène ne pourrait aborder ce type de migration, ce qui nous pousse à réfléchir à une solution d'ordre supérieur.

Solution (contribution) Cette thèse présente (i) Une approche de rétro-ingénierie capable d'extraire des modèles de Microsoft Access. (ii) Un méta-modèle de migration capable de représenter plusieurs langages tout en garantissant qu'un modèle est correct pour une technologie concrète. (iii) Un ensemble de mesures de migration mesurant l'écart technologique entre le système source et la cible prévue. (iv) Des visualisations permettant de se faire une idée du travail nécessaire pour rendre possible une migration architecturale. (v) Une approche itérative interactive de la migration logicielle permettant de migrer vers plusieurs destinations. (vi) Nous fournissons un moteur de migration basé sur des règles permettant : des migrations partielles, basées sur l'application immédiate et différée de règles contextualisées, sur des cibles multiples.

Validation et résultats Pour valider nos solutions, nous menons plusieurs expériences qui donnent des résultats encourageants. (i) Nous validons notre rétro-ingénierie sur 10 projets en mesurant la perte d'information, prouvant que nous pouvons effectuer une migration avec les informations disponibles. (ii) Nous validons notre approche de modélisation sur 34 projets en représentant cinq technologies, et détectons encore des erreurs dans un modèle d'un langage spécifique. (iii-iv) Nous utilisons les métriques et les visualisations pour aider à construire le code source et les plans d'architecture et rédiger les rapports de faisabilité de la migration pour deux grands projets industriels. Nous utilisons également les modèles et

les métriques pour profiler les bibliothèques utilisées afin de guider la construction d'une application de test représentant l'utilisation des bibliothèques sur la base de la règle de Pareto. (v-vi) Nous avons validé la migration d'un formulaire complet vers un back-end Java SpringBoot et un front-end Angular Typescript. Migration de 47 tables et requêtes vers Java et Typescript et de 53 artefacts de bibliothèque vers Java et Pharo.

Mots-clés: Migration des logiciels legataires, Modernisation des logiciels legataires, Transformation du langage, Métriques, Visualisations, Industriel

Contents

1 Introduction **1**
- 1.1 Thesis industrial context 2
- 1.2 Migrating Microsoft Access applications 2
- 1.3 Contributions 4
- 1.4 Structure of the Thesis 6
- 1.5 List of articles 7

2 Background: A Theoretical Framework for Software Modernisation **9**
- 2.1 Building a theoretical framework 10
- 2.2 Software system definitions 11
- 2.3 Legacy systems: The decline of a system 12
- 2.4 Software modernisation 13
- 2.5 Approaches to software modernisation 14
- 2.6 Software migration processes 16
- 2.7 A classification of software migration projects 19
- 2.8 Conclusion 20

3 Microsoft Access to Web technologies: requirements, challenges and constraints **23**
- 3.1 Requirements and challenges of an industrial software migration . 24
- 3.2 An industrial software migration process constraints 30
- 3.3 Shortcomings of software migration:
  Why a holistic approach? 31
- 3.4 Conclusion 34

4 State of the Art: Modelling, Understanding and Transforming **35**
- 4.1 Software migration as iterative reengineering 36
- 4.2 Modelling migrating applications 37
- 4.3 Understanding migrating applications 42
- 4.4 Transforming migrating applications 44
- 4.5 Conclusion 48

I Reverse engineering and Modelling **49**

5 Reverse engineering a Microsoft Access project **51**
- 5.1 Enabling reverse engineering for Microsoft Access 52
- 5.2 Microsoft Access: a partially observable system 52
- 5.3 Component Object Model (COM) technological overview 55

5.4 Mixing static and internal access information 58
5.5 Validation 60
5.6 Threats to validity 68
5.7 Discussion 69
5.8 Conclusion 70

6 Heterogenous Unified Meta-Model: Model migrating systems 71
6.1 Modelling 72
6.2 Understanding language migrations 72
6.3 Heterogeneous Unified Meta-Models 75
6.4 Discussion 79
6.5 Conclusion 80

II Understanding and Measuring Software Migration 81

7 Mind the gap: Understanding Migrating Software 83
7.1 The need to understand 84
7.2 The Alce migration analysis tool 84
7.3 Architectural analysis 85
7.4 Risks and metrics 88
7.5 Visualisations 92
7.6 Alce exemplified on the *ePaie* project 98
7.7 Discussion 101
7.8 Conclusion 102

8 Cases of study: Planning and assessing risk and complexity of software migration 103
8.1 Risk and complexity assessment for the eGRC project 104
8.2 Risk and complexity assessment for the CyclePaie project 107
8.3 Selecting and prioritising the tasks involving library migration for the *ePaie* project 111
8.4 Conclusion 114

III Transforming, Modifying and Producing 117

9 An Interactive, Iterative, Tooled, Rule-Based Migration 119
9.1 Motivations 120
9.2 Overview 120
9.3 Elements of our migration infrastructure 123
9.4 The rule-based context-aware partial migration engine 127
9.5 The approach in action 129
9.6 Discussion 138

9.7 Conclusion 139

10 Case of studies: Migrating language, paradigm, libraries, GUI and architecture from Microsoft Access 141

10.1 Semi-automatic form migration to Typescript Angular front-end and Java Springboot back-end 142
10.2 Migrating tables and queries to a Java back end and a Typescript front end 161
10.3 Automatic library and paradigm use migration from Microsoft Access to Pharo and Java 164
10.4 Conclusion 177

11 Conclusion 179

11.1 Migrating Microsoft Access applications 180
11.2 Problems and Contributions 180
11.3 Meeting the requirements 182
11.4 Future work 185

Bibliography 187

List of Figures

2.1 Solution's Taxonomy Overview (In grey, we find those nodes that are not further explored in this thesis). 13

4.1 Famix meta-model as described by Demeyer *et al.* [Demeyer 1999b] 40

5.1 Access simplified model 54
5.2 Access COM Object model. In white, those elements we can use directly. In grey, those elements depend on other elements and specific interactions. 55
5.3 Architectural inbound design. The image shows the relationship between the different entities and their COM backend. 59
5.4 Validation methodology overview 61
5.5 Completeness Confidence Intervals. 67
5.6 Failures Confidence Intervals. 67
5.7 Original Home screen 68
5.8 Replicated Home screen 69

6.2 Meta-model hierarchy used to represent the Java and Microsoft Access ASG examples. Our entities mainly inherit from Declaration, Reference and Grammatical. They relate using typified relations. 77

7.1 Tool Screenshot: From the left to the right, top to bottom, Toolbar, Project Tree Browser (to navigate the project), Outline (To navigate the elements defined within the selected element), Source Browser (to read and navigate the source code of the selected element). . . 85
7.2 The tagging process: a fragment of the ePaie project being tagged. The figure is split into three parts: 1- Manually tagged, 2- Hierarchical-Propagation Tagged and 3- Usage-Propagation Tagged. 86
7.3 Hierarchically and Usage concerned elements (See text) 87
7.4 Pareto chart schematics. 92
7.5 Class architectural blueprint schematics. The arrows represent an outgoing dependency of a method. We only show arrows leading to another project or another architectural concern. The visualised classes are clustered by architectural concern 93
7.6 Block schematics. (See Text) 94
7.7 Architectural Tangling Highlight schematics: Each block represents a method or function. Blocks are confined within boxes representing their container: class, form, module. Containers are confined in bigger boxes by kind. All is sorted either by ACTI or ACI 95

7.8 Stereotypes for fast reading architectural tangling highlight (See text). 95
7.9 Architectural tangling highlight project. This chart highlights all the architecturally relevant elements of the Agent sub-project. The elements are sorted by ATCI and split by kind: Table, Query, Module, Reports, and Forms. This chart does not contain any Class. . . 97
7.10 Architectural Tangling Highlight: Magnification of the first and more complex Form in the Agent sub-project: Form_FagtVisiteMedicaleSelection. 99
7.11 Dependency Pie chart: All the architectural concerns found in Form: Form_FagtVisiteMedicaleSelection. 99
7.12 Architectural dependency graph applied to Form_FagtVisiteMedicaleSelection. We can observe here a strong dependency on Data Access. Yellow represents GUI. Green represents DataAccess. Purple represents Business, and Red represents Commun. 100
7.13 Analysing fiListeCritere: Dependency complexity metric on Pareto chart. 101

8.1 Study of the complexity of syntactic errors. 104
8.2 Study of the complexity related to the differences in the source. . . 105
8.3 Study of the complexity related to the paradigm shift. 106
8.4 Study of the complexity of the use of dependencies. 107
8.5 Study of the complexity related to grammatical incompatibilities. . 108
8.6 Study of the complexity related to grammatical incompatibilities: contrasting compatible and incompatible entities. 108
8.7 Study of the complexity related to paradigm shift incompatibilities. 109
8.8 Study of the complexity related to grammatical incompatibilities: contrasting compatible and incompatible entities. 110
8.9 Study of the complexity related to dependency usage. 111
8.10 Study of the complexity related to dependency usage. 111
8.11 Analysing *ePaie*: The most used functions. We have to ensure a java replacement of the first 90 functions. 113
8.12 Analysing *ePaie*: The most used members (variables, globals, constants). We have to ensure to have a java replacement for the first 25 members. 114
8.13 Analysing *ePaie*: The most used types. We have to ensure to have a java replacement of the first 30 types. 115

9.1 Migration Process 121
9.2 Microsoft Access model (top left) and Typescript Angular model (bottom right) presented to the user 122

9.3 Rules by target context. On the left side, we find the top levels of the hierarchy of the target ASG. On the right side, we find the installed Productive and Adaptive rules. 132
9.4 Produced “showName” method from “showName” sub-procedure. BinaryOperator and StringLiteral are grammatical nodes; the FunctionInvocation node is not valid in the target model. The numbers in the visualisation refer to the step when the different elements have been created. We note the difference between children and references. 133
9.5 Map directive application: Mapping registering, mappings lookup, adaptive rule lookup. The mapping (ProcedureDeclaration “showName” => MethodDeclaration “showName”) was automatically registered during the Produce directive The numbers in the visualisation refer to the step related. Step 1 creates a mapping. Step 2 looks up all mappings affecting the function invocation “MsgBox”. Step 3 looks up an adaptive. 136

10.1 Form to migrate 142
10.2 Generated Front End 160
10.3 Generated Back-End API request 160
10.4 Bar chart comparing the number of mappings and the impact on parsing and compiling. 175
10.5 Stacked bar chart comparing tests results per language 176

List of Tables

2.1 Articles included in the review 10
2.2 Process x Solution 17
2.3 Classified Articles - Part 1 21
2.4 Classified Articles - Part 2 22

3.1 Requirements table. This table shows multiple approaches and notes if they resolve or not each of the requirements proposed in Section 3.1. The first entry shows the information related to the approach presented in this thesis. 33

5.1 Projects descriptions 61
5.2 Export overview 64
5.3 Query comparison 65
5.4 Table comparison 65
5.5 Form comparison 66
5.6 Report comparison 66

10.1 Mapping examples table 157
10.2 Results of migrating tables and queries to Java and Typescript. . . 163
10.3 Mapping examples table 174
10.4 Translation Results. 174

11.1 This table lists the requirements proposed in Chapter 3. The first shows the information related to the approach presented in this thesis. 184

Chapter 1

Introduction

Contents

1.1 Thesis industrial context 2
1.2 Migrating Microsoft Access applications 2
1.3 Contributions 4
1.4 Structure of the Thesis 6
1.5 List of articles 7

This thesis takes place in an industrial partnership with Berger-Levrault. In an attempt to standardise software development to unify the know-how, reduce costs, and enable the usage of modern architectures, we aim to migrate Microsoft Access applications to web technologies. This study presents the efforts made during the thesis project with this goal.

Section 1.1 contextualises the research project by presenting the Berger-Levrault, the company conducting these modernisation projects, and the nature of our software modernisation. We enumerate other modernisation projects unfolding parallelly in the company. Section 1.2 briefly overviews the Microsoft Access technology and the implications of a modernisation project. Section 1.3 lists the academic, scientific and technological contributions. Section 1.4 provides a blueprint of the thesis. Section 1.5 lists the peer-reviewed articles and reports produced during this project.

1.1 Thesis industrial context

This thesis takes place in an industrial partnership with Berger-Levrault[1]. Berger-Levrault is a 500 years old company which provides software solutions to public and private clients. For example, it builds solutions for managing the electoral roll for the European elections in France, managing city halls' finances, taxes, cemeteries, medical records, teaching solutions, and so on.

A distinctive point is the company's growth strategy. Besides developing new software systems, Berger-Levrault acquires other companies with existing systems in key market segments. The age of the company and the growth strategy yield a company with a rich diversity of working cultures, application domains, programming languages, paradigms, libraries, frameworks, databases, and other artefacts.

Other modernisation projects at Berger-Levrault. In an attempt to standardise software development to unify the know-how, reduce costs, enable the usage of modern technologies, and finally pool the existing solutions into an extensible system using a component-oriented architecture [Allier 2011]. Berger-Levrault started research projects on understanding and proposing a valid system, including Micro-services [Selmadji 2020], micro front-end, software interoperability [Amokrane 2020] and software testing. It also started different projects on Software Evolution and Modernisation projects. Such is the Architectural migration proposed by Zaragoza *et al.* [Zaragoza 2022] and Graphical User Interfaces migration presented by Verhaeghe *et al.* [Verhaeghe 2021a].

The content of this thesis is the record of the work done for the modernisation of Microsoft Access applications, the tooling of the software migration process, and the systematisation and articulation of multiple kinds of migration: language migration, paradigm shift, library migration, architectural migration and GUI migration.

1.2 Migrating Microsoft Access applications

Microsoft Access is a relational database management system (RDBMS) and a fourth-generation language (4GL), comparable with Oracle forms, Visual Fox-pro, and Power Builder. They all aim to ease GUI creation and access to data. Microsoft Access provides a version of Visual Basic for Applications (VBA) as a programming language, an extension of Visual Basic 6. As Visual Basic 6, VBA proposes a hybrid paradigm to tackle down GUI, data storing and processing in a fully controlled and centralised environment. A program developed in Microsoft Access solves problems by orchestrating its first-class citizens: forms, modules, class, tables, queries, reports and macros. From the point of view of object support,

[1] Berger-Levrault: https://www.berger-levrault.com/fr/

VBA is considered object-based [Wegner 1987]. This kind of Rapid Application Development language substantially increases developers' productivity [Subramanian 1996, Beynon-Davies 1999].

1.2.1 Migrating applications.

4GL language implementations, however, provide infrastructure and ease architectural decisions through proprietary language constructions. When migrating to general-purpose languages (such as Java or Typescript), we must consider that many of the language constructions in 4GL languages cannot be represented by the target's language constructions but by libraries, frameworks or even services. This affects multiple parts of the system: architecture, GUI navigation, data access, infrastructure, libraries, frameworks, shipping, deployment, etc.

Most of the infrastructure and architectural decisions proposed by Microsoft Access are incompatible with our technological and strategic target. Berger-Levrault has more than 90 applications written in Microsoft Access, a technology which heavily threatens the decision standardisation and unification of our software base.

Technological targets. Our technological target is web technologies. Moreover, the specific targets for this thesis are Java with SpringBoot for the micro-services-based back-end and Typescript with Angular for the front-end application. However, the approach must consider that these targets may change in the future.

Automatic Migration. Our migration study is expected to yield a solution with a degree of automation, aiming to reduce project costs, including assessment of feasibility and planning and the migration itself.

1.2.2 Problems overview.

We understand the process of migration as an iterative, incremental process split into (i) planning, (ii) understanding what the system to migrate is, (iii) what the technological targets are and what we expect of them, and (iv) transforming the source system into the expected target. Our approach aims to tool the stakeholders of the migration by gathering and organising knowledge and automating transformation.

To do so, we face three families of problems:

Modelling: Any (semi-) automatic approach requires a model. Moreover, the model must support representing multiple technologies and be extracted automatically from the source code. Some questions in this area are:

(i) What is a model able to represent multiple technologies? (ii) What is a reverse-engineering approach to extract models from proprietary technology?

Understanding: We do not directly address planning, but we do address the understanding of the source application and how it is affected by the chosen targets, with different degrees of detail, what can be used in both planning (a strategic step) and understanding (an operational step). Some questions in this area are:

(i) What is the complexity of achieving our target expectations, considering the reality of the source system's source code? (ii) What is the complexity of achieving our architectural expectations, considering the architectural reality of the source system? (iii) What transformation tasks are required to conduct the software migration?

Split and transform: the approach must propose a way to split an application by concern and transform it into multiple technological targets, for which the approach must be technologically agnostic. Some questions in this area are:

(i) What is an approach for splitting a program into multiple targets? (ii) What is an approach for transforming source code into a target language equivalent? (iii) What is an approach for transforming the uses of a library into the uses of another library?

1.3 Contributions

In this section, we present the major contributions of the thesis.

1.3.1 Scientific contributions.

The following contributions are in order of appearance in the problem list, presented in Section 1.2.2.

Modeling

- A reverse engineering approach for extracting models from Microsoft Access.
- A heterogeneous meta-model for representing migrating applications, which reduces the cost of representing new languages and the cost of transformation development.
- A typing ontology that allows understanding when a model is or is not correct according to a programming language.

Understanding

- A suite of metrics measuring the technological gap between a source application and target technologies for complexity and planning assessment.

- Two visualisations helping to understand the architectural complexity affecting the migration of a standalone monolithic application to front-end and back-end applications.

Split and transform.

- A Rule-based interactive iterative approach to software migration with multiple targets. This approach contributes to the *migration of expertise* by guiding the developers to conduct the migration and enable the application's manual *split*.
- A Context-Aware Partial Migration engine based on immediate and delayed Rule application. It fuels the interactive approach and *transforms* a source application into *multiple* technologies.

1.3.2 Technological contributions.

The following technological contributions were required to support and validate the different requirements.

VBAParser [2] A full parser for Visual Basic for Applications, implemented by using Smacc, a parser generator.

Microsoft access extractor [3]

Jindao Microsoft Access COM-based Online model implementation.

JinNS Symbol table for name resolution and the extracting algorithm.

Jindam A graph model that mashes-up information from the COM-Based model and the VBA Parser AST.

Alce Model [4] a Famix meta-model for Microsoft Access applications, and the algorithm for extracting the model out of the Jindao extracting model.

Alce Tool [5] Tool for migration complexity and planning assessment: includes visualisation, tagging algorithm implementations .

Metro [6] Alce model metrics extractor.

Moxing [7] Heterogeneous unified meta-model implementation. It offers support for Microsoft Access, Typescript, Pharo, Java and VisualWorks model.

[2] https://github.com/impetuosa/VBParser
[3] https://github.com/impetuosa/Jindao
[4] https://github.com/impetuosa/Alce
[5] https://github.com/impetuosa/AlcIDES
[6] https://github.com/impetuosa/Metro
[7] https://github.com/impetuosa/Moxing

Spinoza [8] Typing Ontology and its extracting algorithm implementation.

Fylgja [9] Context-Aware Partial Migration engine and the interactive iterative GUI tool implementation.

1.4 Structure of the Thesis

We organised the thesis as follows: Chapter 2 presents the background of software modernisation and migration, introducing some definitions, kinds of migrations, general approaches, and processes. Chapter 3 summarises the requirements of the migration and the migration process. Chapter 4 presents the state-of-the-art software migration measuring, modelling and transforming. We also justify the need for this study by highlighting the shortcoming of existing solutions.

The manuscript is split into three parts. First, in Part I, we present our solutions to reverse engineer Microsoft Access applications and our modelling approach for software migration.

- Chapter 5 explores what a Microsoft Access project is, the threats to reverse engineering inherent to Microsoft Access technology, and how we make a model out of binary representation,
- Chapter 6 shows the Heterogenous Unified Meta-Model. This permissive intermediate representation models the multiple related technologies. It is used to conduct all our transformations and to learn partially and understand the semantic restrictions of each technology automatically through the usage of Typing Ontologies.

Then, In Part II, we present our solutions to understand and measure a Software migration:

- Chapter 7 presents our tool for assessing software migration planning based on metrics and visualisations, along with the definition of metrics and visualisation for our multiple targets. We also present a process of analysis guided by architectural complexity.
- Chapter 8 presents three industrial studies where we used our metrics and visualisations: the risk and complexity assessment for the software migration of (i) eGRC system and (ii) CyclePaie project, and (iii) the task selection and prioritisation process of library migration of the ePaie system.

Then, In Part III, we present our solutions to migrate multiple software aspects to multiple targets:

[8] https://github.com/impetuosa/Spinoza
[9] https://github.com/impetuosa/Fylgja

- Chapter 9 features an Interactive Iterative/Incremental, Tooled, Rule-Based approach to Software Migration, implemented over a Context-Aware Partial Translation engine based on immediate and delayed Rule application.
- Chapter 10 presents three validations of our migration approach: (i) The validation of the iterative, interactive approach to migrating an MS Access form to the front and back end; (ii) The validation of the iterative, interactive approach to migrating MS Access tables and queries to the front and back end and (iii) the validation of the iterative, interactive approach to migrating MS Access library and procedural-paradigm uses to Pharo and Java.

Finally, Chapter 11 summarises and concludes the work presented in this thesis and proposes future work.

1.5 List of articles

1.5.1 Articles on submission

The list of papers on submission in the context of the thesis is listed below in chronological order:

1. Santiago Bragagnolo, Ducasse Stéphane, Anquetil Nicolas and Mustapha Derras. *Interactive, Iterative, Tooled, Rule-Based Migration of Microsoft Access to Web Technologies*. IN SUBMISSION
2. Santiago Bragagnolo, Ducasse Stéphane, Anquetil Nicolas and Mustapha Derras. *Understanding the Migration of Applications with Typing Ontologies*. IN SUBMISSION

1.5.2 Published articles

The list of papers published in the context of the thesis is listed below in chronological order:

1. Santiago Bragagnolo, Abderrahmane Seriai, Stéphane Ducasse and Mustapha Derras. *Risk and Complexity Assessment on the Context of Language Migration*. In International Conference on the Quality of Information and Communications Technology, QUATIC'2021, September 2021
2. Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse, Seriai Abderrahmane and Mustapha Derras. *Analysing Microsoft Access Projects: Building a model in a Partially Observable Domain*. In International Conference on Software and Systems Reuse (ICSR'20), numéro 12541 de LNCS, December 2020

3. Santiago Bragagnolo, Benoît Verhaeghe, Abderrahmane Seriai, Mustapha Derras and Anne Etien. *Challenges for Layout Validation: Lessons Learned*. In International Conference on the Quality of Information and Communications Technology, QUATIC'2020, September 2020

1.5.3 Technical reports

We also produce technical reports related to the study.

1. Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse, Abderrahmane Seriai and Derras Mustapha. *Software Migration: A Theoretical Framework (A Grounded Theory approach on Systematic Literature Review)*. Rapport technique, Berger-Levrault and Inria Lille Nord Europe, 2021

2. Santiago Bragagnolo, Ducasse Stéphane, Anquetil Nicolas, Abderrahmane Seriai and Mustapha Derras. *Alce: Predicting Software Migration*. Rapport technique, Inria, January 2023

3. Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse and Derras Mustapha. *Reporting Context Aware Partial Translation engine based on immediate and delayed Rule application*. Rapport technique, Inria, December 2022

CHAPTER 2

Background: A Theoretical Framework for Software Modernisation

Contents

2.1 Building a theoretical framework 10
2.2 Software system definitions 11
2.3 Legacy systems: The decline of a system 12
2.4 Software modernisation 13
2.5 Approaches to software modernisation 14
2.6 Software migration processes 16
2.7 A classification of software migration projects 19
2.8 Conclusion 20

This chapter aims to clarify a system's different parts, describing the decline affecting them and how this makes an industrial system into a legacy system. These different parts do not explain why migration happens but explain what part is essential in a specific kind of migration.

Section 2.1 presents briefly the methodology used to build the content of this chapter. Section 2.2 introduces some basic definitions that may have multiple interpretations to ensure understanding. Section 2.3 defines a legacy system in terms of two agents of decline: decadence and obsolescence. Section 2.4 introduces a taxonomy for *Software Modernisation* solutions. Section 2.5 introduces a taxonomy of the families of approaches existing in the literature. Section 2.6 introduces a taxonomy of the families of processes existing in the literature. Section 2.7 classifies the articles used to build up this chapter based on the different taxonomies introduced by the chapter. Section 2.8 concludes the chapter.

2.1 Building a theoretical framework

To build this chapter, we ran a systematic literature review published as a report [Bragagnolo 2021a], available in https://hal.inria.fr/hal-03171124v2. We do not include the experiment details and methodology in this chapter, as it would add many pages to the thesis. However, we include Table 2.1 with the different articles in the review, as we classify them in the end with the different extracted taxonomies, to give an idea of the different uses of different approaches. The table includes two sets of articles, those extracted by the systematic review process (from 1 to 30) those added as support for definitions (I to VII).

Table 2.1: Articles included in the review

#	Year	Title	Publisher
1	2019	GUI Migration using MDE from GWT to Angular 6: An Industrial Case [Verhaeghe 2019]	IEEE
2	2018	An Approach for Creating KDM2PSM Transformation Engines in ADM Context: The RUTE-K2J Case [Angulo 2018]	ACM
3	2017	White-Box Modernization of legacy Applications [Garcés 2017]	Springer
4	2016	A Survey on Survey of Migration of legacy systems [Ganesan 2016]	ACM
5	2015	Modernization of legacy systems: A Generalized Roadmap [Jain 2015]	ACM
6	2014	How do professionals perceive legacy systems and software modernization? [Khadka 2014]	ACM
7	2014	A framework for architecture-driven migration of legacy systems to cloud-enabled software [Ahmad 2014]	ACM
8	2013	Migrating legacy Software to the Cloud with ARTIST [Bergmayr 2013]	IEEE
9	2012	Seeking the ground truth: a retroactive study on the evolution and migration of software libraries [Cossette 2012]	ACM
10	2012	Searching for model migration strategies [Williams 2012]	ACM
11	2012	A lean and mean strategy for migration to services [Razavian 2012]	ACM
12	2010	Extreme maintenance: Transforming Delphi into C# [Brant 2010]	IEEE
13	2009	Parallel iterative reengineering model of legacy systems [Su 2009]	IEEE
14	2008	Can design pattern detection be useful for legacy system migration towards SOA? [Arcelli 2008]	ACM
15	2008	Developing legacy system migration methods and tools for technology transfer [De Lucia 2008]	Wiley & Sons
16	2007	OPTIMA: An Ontology-Based PlaTform-specIfic software Migration Approach [Zhou 2007]	IEEE
17	2007	Reversing GUIs to XIML descriptions for the adaptation to heterogeneous devices [Di Santo 2007]	ACM
18	2005	Quality driven software migration of procedural code to object-oriented design [Zou 2005]	IEEE
19	2004	Incubating services in legacy systems for architectural migration [Zhang 2004]	IEEE
20	2003	Network-centric migration of embedded control software: a case study [de Souza 2003]	IBM Press
21	2002	C to Java migration experiences [Martin 2002a]	IEEE
22	2002	A framework for migrating procedural code to object-oriented platforms [Zou 2001]	IEEE
23	2000	A Survey of legacy system Modernization Approaches [Comella-Dorda 2000]	DTIC [1]
24	1998	Code migration through transformations: an experience report [Kontogiannis 1998]	IBM Press
25	1997	Lessons on converting batch systems to support interaction: experience report [DeLine 1997]	ACM
26	1997	Reverse engineering strategies for software migration (tutorial) [Müller 1997]	ACM
27	1996	Strategic directions in software engineering and programming languages [Gunter 1996]	ACM
28	1996	Rule-based detection for reverse engineering user interfaces [Moore 1996]	IEEE
29	1995	Workshop on object-oriented legacy systems and software evolution [Taivalsaari 1995]	ACM
30	1994	Knowledge-based user interface migration [Moore 1994]	IEEE
I	2015	ISO IEC 90003 (ISO 9001 applied to Software) [ISO 2015]	ISO
II	2011	ISO IEC 25010 (ex ISO IEC 9126) [ISO 2011a]	ISO
III	2011	ISO IEC 42010 [ISO 2011b]	ISO
IV	2006	ISO IEC 14764 [ISO 2006]	ISO
V	2002	Object-Oriented Reengineering Patterns [Demeyer 2002]	M Kaufmann
VI	1990	Reverse Engineering and Design Recovery: A Taxonomy [Chikofsky 1990]	IEEE
VII	1985	Program evolution: Processes of software change. [Lehman 1985]	LAP [2]

2.2 Software system definitions

System. Following the definition given by [ISO 2011b], it is a man-made entity that may be configured with one or more of the following: hardware, software, data, humans, processes (e.g., processes for providing service to users), procedures (e.g., operator instructions), facilities, materials and naturally occurring entities. We also add that all these entities and their relationships configure what we understand as the environment where our software takes place.

Architecture & Design. Following the definition given by [ISO 2011b], we recognise architecture as the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and principles of its design and evolution. Its elements: the constituents that make up the system; the relationships: both internal and external to the system; the principles of its design and evolution. Furthermore, we differentiate architecture from design: **architecture is outwardly focused** on the system in its environment, whereas **design is inwardly focused** once the system boundaries are set.

From this, we can infer that architectural evolutions also entail the evolution of the inward design, directly impacting the piece of software and indirectly impacting the implementation of the business rules.

Software Quality. According to [ISO 2011a] we talk about quality from three points of view. Developers perceive the quality: (i) "internally" by measuring the source code quality and metrics, documentation, and knowledge of the maintaining organisation; (ii) "externally" by measuring its artefact behaviour. Users perceive the quality as "in-use" as the software's capacity to accomplish requirements and adapt to new changes. [ISO 2011a] also spots the inter-relationship of these qualities.

Software Modernity. The modernity of software is related to the distance between the up-to-date techniques and technologies of software development and those used during the development of the source code. An example is if this software cannot profit from using up-to-date technologies and concepts, such as AI, IoT, Blockchain, and microservices.

Software Continuity. The continuity of a piece of software (persistence or permanence) is directly related to the resource allocation policy for its maintenance and evolution. Despite the modernity or the quality, software continuity is related to how much this software is needed and how many resources the owners can afford to keep it working. A direct implication of continuity is incrementing the investment value in multiple aspects: money, time, and knowledge. Lehman *et al.* [Lehman 1985] proposes *the law of continuing change*: A program that is used

in a real-world environment *must* change or become progressively less useful in that environment.

2.3 Legacy systems: The decline of a system

The constant passage of time and evolution often contribute to a system's decline. In our context, we recognise two main kinds of decline: (i) the decadence, (ii) the obsolescence.

Decadence. Decadence is the continuous deterioration of the *inherent internal qualities* of a software: unreliable documentation, lack of knowledge, increased accidental complexity, highly tangled and coupled source code, and loss of consistency and cohesion. The decadence of the system *hampers its evolution*. [Kontogiannis 1998] states a significant fact on this aspect: "Some components of the system are not owned by any member of the development team and are therefore very difficult to maintain. Not surprisingly, the team is reluctant to perform radical changes to its structure since this may negatively affect its overall performance.". Decadence is visible in different internal artefacts and aspects of a system: design, division of concerns (UI, Data, business logic, etc.), the used API, the language and paradigms (as they directly impact the source code), and the source code itself.

Obsolescence. For obsolescence, we understand the changes in the environment where our software exists and how these changes affect the *inherent external qualities* of the software: the apparition of new technologies and paradigms, or the deprecation of dependent technologies impacts the way a system interacts with other systems: Apparition of online services competition, radically cheaper infrastructure, the deprecation of dependent software (libraries, compilers, etc.), the out-of-production of required hardware platforms, changes in business legislations, etc. The obsolescence of the system *justifies and causes its evolution*. [de Souza 2003] exposes the urgency of system evolution in the context of a project that requires enabling network communication on a system that includes embedded software since this requirement implies hardware-level modifications. The obsolescence is visible in different external constructs of a system: the architecture, the runtimes, the hardware, GUI, and the third-party artefacts: libraries, SDK, and frameworks.

Legacy systems. These are successful systems with a long continuity, which struggle to accomplish new strategic decisions due to some grade of *decadence* or *obsolescence* at some part of the system. [De Lucia 2008] spots the importance of systems that runs 24/7. [Kontogiannis 1998] points out that software that migrates "are often mission-critical for the organisation that owns and operates them". One of the interviews in Khadka *et al.* [Khadka 2014] proposed a definition: "My definition of a legacy system is systems and technologies that do not belong to your

strategic technology goals". This is a weak definition, but it points out something important: a system can become a legacy with a simple strategic change. Demeyer *et al.* [Demeyer 2002] says that a legacy system is a constantly evolving system critical to your business and cannot be upgraded or replaced except at a high cost. The constant evolution of this system is what exposes it to decline.

We propose to recognise the kind of legacy system in terms of how it is affected by decadence or obsolescence. (i) legacy system due to third-party library obsolescence, (ii) legacy system due to an obsolete programming language, (iii) legacy system resulting in decadent source code, (iv) legacy system due to decadent design.

2.4 Software modernisation

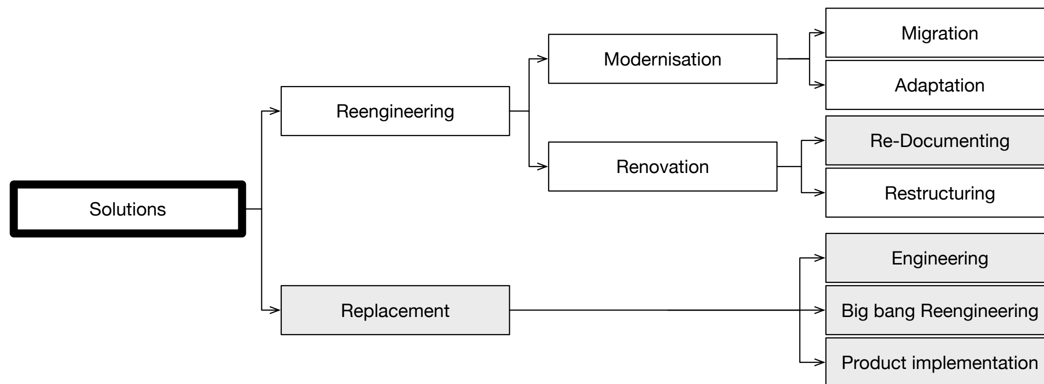


Figure 2.1: Solution's Taxonomy Overview (In grey, we find those nodes that are not further explored in this thesis).

Reengineering. Chikofsky *et al.* [Chikofsky 1990] proposes reengineering to be the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

Modernisation. We name Modernisation all processes to recover a system from *obsolescence*, achieving better integration with the environment and enhancing the external quality of the system. These processes affect the external and internal elements of a legacy system.

Adaptation is a modernisation process that enables the usage of a new technological environment without threatening currently used technology. There are many kinds of adaptations, from, e.g., (i) [Gunter 1996], proposing to compile C in C++, to be able to add new code in an object-oriented fashion, too, e.g., (ii) [de Souza 2003] proposing to modify hardware, or, e.g., [Di Santo 2007] who adapts a website to be rendered on different running devices.

Migration is all Modernisation process that moves from one Source technological environment to a Target technological environment that is in relation of mutual exclusion (either for technological or strategical reasons) with the Source environment. There are many kinds of migrations, like source code translation proposed by [Brant 2010, Kontogiannis 1998, Martin 2002a], GUI migrations presented by [Verhaeghe 2019, Garcés 2017, Moore 1994], library migration [Zhou 2007, Cossette 2012, Martin 2002a] or architectural migration [Zhang 2004, Razavian 2012, Bergmayr 2013, De Lucia 2008, de Souza 2003]

Renovation. We name Renovation all processes that recover a system from *decadence*, achieving better internal quality or understanding of the internal structure. These processes affect only internal elements of a legacy system.

Restructuring is all Renovation processes issued over the source code (e.g., refactoring).

Re-Documenting is all Renovation process that produces new or enhances existing documentation of the code, such as writing manuals, specifying processes, and formalising requirements. "The spectrum of reengineering activities includes re-documentation, restructuring of source code, the transformation of source code, abstraction recovery, and reimplementation." [Müller 1997]

Replacement. Replacement is all processes that discard the existing system and establish a different one.

Engineering is a Replacement process that creates a new system based on understanding the current requirements.

Big-bang Reengineering is all Replacement processes that create a new system based on understanding the historical requirements by reverse engineering an existing design. Proposed and rejected by many of the articles, such as Brant *et al.* [Brant 2010].

Product implementation is all Replacement processes that implement and customise a Commercial Off-The-Shelf (COTS) system to solve the current requirements. For example, De Souza *et al.* [de Souza 2003] proposes the possibility of an off-the-shelf product.

Summary. In this section, we split the kinds of solutions according to what they directly affect and how they solve the problem. This is done for the sake of understanding. It is essential to understand that software modernisation is often about the overlapped application of all these techniques. To achieve a proper refactor, we may need to update a library, which would entangle a modernisation of the source code. We will likely need to modify our software's API to migrate to a target architecture, impacting the inbound and business logic design.

2.5 Approaches to software modernisation

In our study, we found three principal families of technical approaches that tackle most of the reengineering challenges in our field. They are those based on a deep understanding of the Source system/subsystem (white-box approaches), those based on the analysis of input and outputs [Comella-Dorda 2000] (black-box approaches) and those based on hybrid approaches (grey-box approaches).

2.5.1 Black-box Approaches

Black-box or external approaches are named after the fact that they disregard the internal composition of the system and focus on understanding the inputs and outputs of a legacy system within an operating context to gain an understanding of the system/subsystem interfaces. These approaches often imply low or no modifications to the existing system. Black-box strategies are often based on wrapping techniques.

Wrapping. consists of surrounding a piece of software with a software layer that hides unwanted complexity and exports a new interface. Wrapping removes mismatches between the interface exported by a software artefact and the interfaces required by current integration practices. Since wrapping impacts over devices aiming to enable communication, it is only applicable on the different levels of interoperability: Third-party solutions, exhibited API/ABI, and Architecture.

2.5.2 White-box Approaches

White-box or internal approaches are named after the fact that they consider the internal composition of the system. Often based on an initial reverse engineering process to gain a deep internal understanding of the Source system/subsystem. This process usually aims to identify components and relationships at different levels of abstraction (classes, patterns, dependencies, etc). Automatic and semi-automatic white-box techniques are usually based on producing symbolic models, such as meta-models or ontologies. These approaches often imply a high amount of modifications to the existing system. White-box approaches are often based on transforming techniques.

Transforming. produces a software component semantically equivalent to an existing one. This produced software component responds to an equivalent level of abstraction and exhibits different technological features or assumptions. Since a transformation impacts directly or indirectly the source code, it can be applied to all the different internal and external parts of the software. Architecture, Design, Language, exhibited and used API/ABI, Paradigm, Deployment environment, and Third-party products.

2.5.3 Grey-box Approaches

Grey-box or hybrid approaches are those approaches that (i) use white-box approaches for enabling certain granularity on black-box approaches or (ii) use general black-box approaches to reduce risks and not operational time of invasive white-box approaches.

On the first kind, we find most of the proposals of migration of software to service architectures using internal approaches to recognise parts of a system and decompose it, enabling to wrap parts of a system instead of the whole system [Ganesan 2016].

We found that the second approach, especially on modernisation processes, required delegating what once was a concern of the system to a third-party product. Such is the case of the migrations from language-support data management to third-party products (most of the iconic cases come from the migration from COBOL registry files to RDBM systems) [De Lucia 2008].

2.6 Software migration processes

Software migrations are often lengthy and highly risky enterprises [Razavian 2012, Khadka 2014]. Such projects often deal with legacy systems that suffer from both Decadence and Obsolescence on multiple artefacts.

In short such projects are bound to a lot of detailed variables that impose the instrumentations of many times ad-hoc processes, which makes it especially hard (if not impossible) to generalise practical procedures (as the suitable process we understand an exhaustive definition able to fit all possible cases of modernisation and renovation), but only some process form for the sake of knowledge organisation.

According to our study of the literature, we recognise that, in general, software migration responds to two families of processes: Attached and detached, relative to the target project.

Detached processes. *Detached* process often responds to variations of a *phased* process or the butterfly method proposed by Wu *et al.* [Wu 1997]. This model is related to processes that take as input a system and give as output a new system that should comply with the old and new specifications. [Williams 2012, Ahmad 2014, Jain 2015, Bergmayr 2013]. We name this kind of process *detached* because the process does not relate to the target any other way than generating it. The produced target is not used as input.

Disadvantages: Due to the forking nature of the process (which produces a new system), it threatens the maintenance and development of new features. This process requires producing a new system based on the original system [Bianchi 2003, Wu 1997]. This is often split into parts like a first-class citizen: classes, modules, widgets, etc. This kind of granularity imposes the entity over the feature, requiring

the process of whole entities to produce a feature; this is likely to increase the delivery time. Products may take much time to be implemented, seen and valorised.

Advantages: On the other hand, it does not threaten the quality or stability of the Source system. The Source system can still be used as it is [Bianchi 2003, Wu 1997].

Attached processes. These processes respond to variations of the classical Spiralling forward-engineering model [ISO 2006] or the chicken little method proposed by Brody *et al.* [Brodie 1995]. Related to the nature of a process that takes as input a system and gives as output the same system but modified. [Zhang 2004, DeLine 1997]

Disadvantages: Due to the continuously integrating nature of the process, it is essential to remark that it threatens the stability and internal consistency of the system [Bianchi 2003, Brodie 1995].

Advantages: Each iteration of the process may apply transformations of arbitrary size [Bianchi 2003, Brodie 1995]. The smaller the size of a modification over a running system, the easier to test, deliver and deploy new versions. Regular delivery increases the access to user feedback and the visibility and valorisation of the migration process.

Table 2.2: Process x Solution

	Modernisation		Renovation
Process	Migration	Adaptation	Restructuring
Detached	White / Grey-box	Not found	Refactoring / Transform
Attached	Black / Grey-box	White / Black-box	Refactoring / Transform

As shown in Table 2.2, we find that migration responds to attached and detached processes, Adaptation, in our findings, responds only to Attached processes, as it is about adding support to a new feature. On the renovation side, we find both kinds of processes. Below we present each step.

2.6.1 Process steps

Following, we present the generic steps in a both attached and detached processes.

Plan. Activities in this step are generally conducted to define the reach and expectations of the process at the operational level [ISO 2015], including risk and feasibility assessment.

Müller *et al.* [Müller 1997] recognises that risk is related to planning ”Minimising the migration risk is a key requirement. The most common strategy is an incremental approach to minimise the risk”.

Razavian *et al.* [Razavian 2012] remarks the importance of understanding "Associating costs and risks to core activities makes the core an even more powerful tool for planning how to migrate."

Understand the source System. Activities in this step are typically conducted to acquire knowledge of the system. [Ahmad 2014, Bergmayr 2013]. These activities are accomplished manually, semi-automatically, or automatically.

The proposed activities range from (i) intellectual understanding (based on interviewing team members of the project, reading documentation and or code [Razavian 2012]), to (ii) computational models built from reverse engineering (as those proposed especially by model-driven engineering [Angulo 2018, Verhaeghe 2019, Garcés 2017, Williams 2012, Brant 2010]) or (iii) ontological methods [Zhou 2007], that propose a computational representation of the semantics and structures of the system.

This knowledge is required at many levels, from management and planning (to measure risk, to prioritise tasks, etc. [Razavian 2012, Cossette 2012]) to the input of automatic/semi-automatic algorithms with many usages such as code enhancement recommendations, language translation, etc. [Williams 2012, Brant 2010].

Understand the expectations of the target System. This step's activities are usually conducted to acquire knowledge of the Target system. [Ahmad 2014, Bergmayr 2013]. These activities usually are accomplished manually. The proposed activities are related to understanding how the new system will behave and interact with the environment. This knowledge is required to choose a correct and optimal approach [Ahmad 2014] for the process, estimating costs, times, risks, and assessing task prioritisation [Razavian 2012, Cossette 2012].

Transform knowledge. Activities in this step are normally conducted to work over the acquired knowledge regarding the process expectations. [Ahmad 2014] These activities are accomplished manually, semi-automatically, or automatically. The nature, size, and order of the tasks change from white to black-box approaches. Still, these activities range from the intellectual understanding (of the required transformations and re-structuration to apply to accomplish the target expectations of the current process as proposed by Razavian *et al.* [Razavian 2012] to leverage and transform computational models built during the previous step, to fit better on the Target system restrictions [Moore 1994, Arcelli 2008], or [Zou 2001] who uses clustering algorithms over models for proposing classes and methods in the context of procedural to object-oriented migrations).

Modify system. Specific for Attached Process processes. This step's activities are usually conducted to apply the transformed knowledge to the current system. These activities are accomplished manually, semi-automatically, or automatically.

The nature of the modification range from modifying some asset of the system (source code, documentation, etc.) manually [Razavian 2012, Arcelli 2008, DeLine 1997] to the automatic/semi-automatic modification of these assets [Zhou 2007].

Produce destination. Specific for Detached Process processes. Activities in this step are typically conducted to use the transformed knowledge to produce a Target system. These activities are accomplished manually, semi-automatically, or automatically. The nature of the product range from the manual creation of the Target system (based on the transformed knowledge), to the automatic/semi-automatic generation of this Target system [Williams 2012, Garcés 2017, Angulo 2018]

2.7 A classification of software migration projects

In Table 2.3 and Table 2.4, we present all the reviewed articles along with their classification. The columns offer the following information:

Article ID: the entry of the article in Table 2.1.

Legacy System: A brief description of the legacy system the classified article addresses.

Main driver: Different drivers are explained in detail in our report [Bragagnolo 2021a]. To briefly define it, a driver is a strategic objective behind the migration project.

Main Objective: Different objectives are explained in detail in our report [Bragagnolo 2021a]. To briefly define it, a driver is a tactical decision: what is done to address the strategic objective technologically.

Solution Kind: The kind of solution, following the taxonomy proposed in Section 2.4: migration, adaptation or both.

Approach Kind: The approach kind, following the taxonomy proposed in Section 2.5: black-box, white-box or grey-box.

Approach: The approach or the kind of technological operation, explained in Section 2.5: transformation or wrapping.

Process: The kind of process used by the approach, as defined in Section 2.6: detached or attached.

We note that some articles either do not apply (*N/A*) to a specific column or exhibit values such as *Many* or *All*. This is related to the nature of the article. Some articles do not pursue a real industrial case but propose a general approach to solve something, *e.g.* architectural migration [Bergmayr 2013], for which they have no legacy system. Other articles propose systematic literature reviews [Comella-Dorda 2000], for which they have more than many solutions or processes. Finally, other articles analyse the perception and expectations of a software migration [Khadka 2014, Razavian 2012], which also impact the values on their columns.

2.8 Conclusion

In this chapter, we started by agreeing on some definitions (Section 2.2). This is required to understand the rationale behind the taxonomies and this thesis. We discussed legacy systems and the concepts of decadence and obsolescence (Section 2.3). We presented a solutions taxonomy based mainly on renovation and modernisation (Section 2.4). We presented a taxonomy of approaches to software modernisation. We categorised them as the black-box, white-box, and grey-box approaches, presenting the main technological approaches: wrapping and transformations (Section 2.5). We also described the two main kinds of processes in software migration according to their relation to the project's life cycle (Section 2.6). We classified each reading with the proposed taxonomies (Section 2.7).

With this overview and framework, we are ready for the Chapter 3, where we analyse the different requirements and challenges of this industrial migration, and how those make the presented solution fall short.

Table 2.3: Classified Articles - Part 1

Article ID	Legacy System	Main driver	Main Objective	Solution Kind	Approach Kind	Approach	Process
1 [Verhaeghe 2019]	GWT Web application	Move from dying technology	UI Translation	Migration	White-box	Transformation	Detached
2 [Angulo 2018]	N/A	N/A	KDM to PSM transformation	Migration	White-box	Transformation	Detached
3 [Garcés 2017]	Oracle forms application	Moving from dying technology	UI Translation	Migration	White-box	Transformation	Detached
4 [Ganesan 2016]	N/A	Many	Many	Migration	All	All	All
5 [Jain 2015]	N/A	Enable new architectural variables	Migrate To Service	Migration	All	All	All
6 [Khadka 2014]	Many	Many	N/A	N/A	N/A	N/A	N/A
7 [Ahmad 2014]	N/A	Enable new architectural variables	Enable Cloud	Migration	Grey-Box	Wrapping	Detached
8 [Bergmayr 2013]	N/A	Enable new architectural variables	Enable Cloud	Migration	Grey-Box	Wrapping	Detached
9 [Cossette 2012]	N/A	N/A	Library Migration	Migration	White-box	Transformation	N/A
10 [Williams 2012]	Object Model	N/A	N/A	Adaptation	White-box	N/A	Detached
11 [Razavian 2012]	N/A	Enable new architectural variables	Migrate To Service	Migration	Grey-box	Wrapping	Detached
13 [Brant 2010]	Delphi application	Move from dying technology	Translation	Migration	White-box	Transformation	Detached
14 [Su 2009]	N/A	N/A	N/A	All	N/A	N/A	N/A
15 [Arcelli 2008]	Object oriented application	Enable new architectural variables	Migrate To Service	Migration	N/A	N/A	N/A
16 [De Lucia 2008]	Cobol application	Enable New Business / Markets	Migrate to service	Migration	Grey-box	Wrapping	Detached

[N/A] Not applies

[All] All the options of the taxonomy are to be found in this article

[Many] More than one option of the taxonomy is to be found in this article

Table 2.4: Classified Articles - Part 2

Article ID	Legacy System	Main driver	Main Objective	Solution Kind	Approach Kind	Approach	Process
17 [Zhou 2007]	C/C++ application	Move from dying technology	Library Migration	Migration	White-box	Transformation	Detached
18 [Di Santo 2007]	Java AWT Application	Enable new features	GUI Migration | GUI Adaptation	Adaptation after Migration	White-box	Transformation	Detached
19 [Zou 2005]	Procedural Application	Enable new features	Paradigm change	Migration	White-box	Transformation	Detached
20 [Zhang 2004]	C/C++ Application	Enable new architectural variables	Migrate To Service	Migration	Grey-box	Wrapping	Detached
21 [de Souza 2003]	Embedded system	Enable new features	Adapt embedded system to support networking	Adaptation	White-box	Transformation	Attached Process
22 [Martin 2002a]	C application	Enable new features	Translation	Migration	White-box	Transformation	Detached Process
23 [Zou 2001]	Procedural Application	Enable new features	Paradigm change	Migration	White-box	Transformation	Detached
24 [Comella-Dorda 2000]	N/A	Many	Many	All	Black-box	Wrapping	All
25 [Kontogiannis 1998]	PL/IX Application	Move from dying technology	Translation	Migration	White-box	Transformation	Detached
26 [DeLine 1997]	Batch application	Enable new features	Adapt batch to support interactive control	Adaptation	White-box	Transformation	Attached
27 [Müller 1997]	N/A	N/A	N/A	Migration	Black-box	Wrapping	N/A
28 [Gunter 1996]	N/A	N/A	Paradigm change	Migration	N/A	N/A	N/A
29 [Moore 1996]	Texte UI Application	Enable new features	UI Translation	Migration	White-box	Transformation	Detached
30 [Taivalsaari 1995]	N/A	N/A	N/A	All	N/A	N/A	N/A
31 [Moore 1994]	GUI Application	Move from dying technology	UI Translation	Migration	White-box	Transformation	Detached

[N/A] Not applies
[All] All the options of the taxonomy are to be found in this article
[Many] More than one option of the taxonomy is to be found in this article

CHAPTER 3

Microsoft Access to Web technologies: requirements, challenges and constraints

Contents

3.1 Requirements and challenges of an industrial software migration 24

3.2 An industrial software migration process constraints 30

3.3 Shortcomings of software migration: Why a holistic approach? 31

3.4 Conclusion 34

This chapter aims to help the reader to understand the shortcomings of the state of the art presented in Chapter 4. To do so, this chapter introduces the challenges and constraints of the industrial software migration of Microsoft Access to web technologies.

Section 3.1 presents the kind of migrations involved in this industrial case. Section 3.2 presents the constraints of the migration process. Section 3.3 explains the shortcoming of the software migration literature and offers a glimpse of the approach presented in this thesis.

3.1 Requirements and challenges of an industrial software migration

Our migration project has the difficult task of migrating a rich fourth-generation programming language (4GL) to multiple general-purpose languages responding to new architectural designs.

This section presents different aspects of this industrial case of software migration. We highlight the requirements and challenges of each aspect. We detail the challenges and requirements for automatic migration.

This section presents the different aspects of this migration project, express them as requirements and presents the main challenges this thesis will address.

3.1.1 A technologically agnostic approach.

Our project involves a minimum of three technologies. One of them is logically fixed: the source system technology. The targets we investigate are Java and Typescript; however, the target decision can change in the future.

- Requirement: The approach must be technologically agnostic.
- Challenge: All the different aspects of the migration approach must apply to other technologies.

3.1.2 Programming language migration

This migration of source code as translation requires multiple considerations.

Translate and transform to multiple targets. Microsoft Access projects support two kinds of source code: The “macros” language – a specific point-and-click language for Microsoft Access – and VBA, a language inspired by Visual Basic, adapted for use in the context of Microsoft Access. Our targets are expected to be Java and Typescript, but this decision can change in the future.

- Requirement: The approach must translate and transform code to the expected targets: Java, Typescript and HTML.
- Challenge: To translate two different sources into two targets with different requirements using the same approach.
- Challenge: To translate incompatible structures [Brant 2010,Kontogiannis 1998]. VBA actively supports and encourages the usage of language structures that are undesirable or have no equivalent in our target languages, such as GO-TO. Java and Typescript use more modern control flow structures. Such as try/catch for error management.

Explicity dependency declaration. Microsoft Access stores all the code into a single binary file. Furthermore, VBA does not support namespaces or packages. All the module functions and classes defined in a project are visible within the project.

- Requirement: The target source code must include explicit dependencies declaration (by using import).
- Challenge: To provide explicit import clauses for both targets for the translated code.

3.1.3 Library and infrastructure migration

When translating source code to a target platform, we must consider the available artefacts and how to use them.

Libraries migration. Microsoft Access uses different kinds of libraries. Some require the Microsoft Access runtime; others also work in .Net technologies. None is available in Java with Springboot and Typescript with Angular. Moreover, while VBA supports static, dynamic and hybrid typing, Java and Typescript are statically typed.

- Requirement: The target systems must use target libraries [Kontogiannis 1998, Brant 2010, Martin 2002a, Trudel 2012].
- Challenge: To support the definition of custom mappings between source and target libraries.
- Challenge: To provide ways to ease the library migration when no equivalent is available.
- Challenge: The target source must use the target types correctly [Terekhov 2000].

Infrastructure migration. Besides the fact that the projects use multiple third-party libraries (which must be replaced in the target system), VBA language provides an SDK with seamless language integration and includes basic types and infrastructure.

For example, in Microsoft Access, the development of graphical interfaces is done using a wizard (drag and drop). The wizard adds graphical controls to the GUI built. These controls and forms can be *declaratively* and *transparently* bound to a database table.

Our targets do not offer this kind of service out of the box. They rely on frameworks and libraries.

- Requirement: The target systems must be built by understanding the *behind the scenes* of Microsoft Access and provide similar infrastructure by using target libraries and frameworks [Kontogiannis 1998, Brant 2010, Martin 2002a, Trudel 2012].

- Challenge: To provide ways to interpret infrastructure usage.

- Challenge: To generate target code based on the usage of libraries or frameworks.

3.1.4 Paradigm migration

We must consider the available concepts when migrating the design of a source code to a target platform.

Procedural to object-oriented migration. VBA supports procedural and object-based paradigms. Java exclusively supports object-oriented constructions, while TypeScript supports procedural and object-oriented programming (OOP); In the case of java, static methods in static classes can be used to simulate procedural programming.

however, our target is to use Angular, a framework based on component-oriented development.

- Requirement: The translated code must be expressed in terms of classes, methods and objects [Zou 2001, Kontogiannis 1998].

- Challenge: To provide strategies to translate modules, variables, functions and sub-procedures to OOP. This may imply clustering functions into classes, knowing that no automatic approach has proved definitive.

- Challenge: To provide strategies to translate the usage of variables, functions and sub-procedures (invocations) to OOP. This requires being able to infer a "receiver".

Microsoft Access to object-oriented migration. Besides the Modules and Class-Modules, Microsoft Access offers other First-Class-Citizen (FCC): Forms, Reports, Tables and Queries. For our goals, Java and Typescript only provide classes.

- Requirement: The translated FCC must be expressed in terms of classes, methods and objects [Zou 2001, Kontogiannis 1998].

- Challenge: To provide equivalent representations in both targets for Forms, Reports, Tables and Queries, without losing semantics on the process [Terekhov 2000].

3.1.5 Architectural migration challenges

We must consider the assumptions and rules of an architectural target when migrating architecture.

Standalone to Network-centric migration. Microsoft Access applications are so-called "standalone" applications, *i.e.* they are developed to be deployed centrally, although they have the ability to interact with remote data servers. Furthermore, a standalone application can access the resources of its deployment environment (user computer), such as the operating system used, the file system, printers, etc. A web application rarely accesses users' resources but network resources, assuming the existence of a network and shared resources. These two environments' differences invalidate many original development assumptions, requiring adaptation or redevelopment during migration [de Souza 2003].

This implies that multiple parts of the source system deal with entities that just *do not exist* on the target.

- Requirement: Migrate only the pieces of software needed on the target [Brant 2010].
- Challenge: Different target systems must be built from specifically selected pieces of the source system. This implies supporting the splitting of software.
- Challenge: Considering that many pieces we migrate to the targets rely on parts we are not migrating implies that we must cope with partial migrations.

Monolithic to front-end and microservices backend migration. Our industrial systems are monolithic. A monolithic system is often described as a single-tier system in which the user interface, business logic and data layer are combined into a single application. Microservices are one of the latest trends in software development that has emerged from service-oriented architecture styles. Microservices are expected to specialise in specific business and technological concerns, small, highly cohesive, loosely coupled services, each independently deployable and communicating with mechanisms such as REST or a message bus like RabbitMQ. The contrast between these two architectures is dramatic. The source code has been developed assuming local synchronised execution with low-level shared resources, such as memory and stack.

- Requirement: To split the source system into front-end and many possible back-end targets [De Lucia 2008, Zaragoza 2022, Zhang 2004, Razavian 2012].
- Requirement: To establish communication between the different target artefacts. [De Lucia 2008, Zaragoza 2022, Zhang 2004, Razavian 2012].

- Challenge: Different target systems must be built from specifically selected pieces of the source system. This implies supporting the splitting of software.
- Challenge: Splitting a monolithic application into multiple *interacting independent applications* entails transforming multiple local calls into remote calls. This transformation also implies the definition of an API. This could also imply modifying the code to make it asynchronous.

3.1.6 Migrating the GUI

Graphical user interfaces for Microsoft Access applications are developed using Microsoft Office GUI components (see also Section 3.1.3). These components generally use the libraries provided by the Microsoft Windows operating system. The graphical interfaces use fixed layouts and are built by point-and-click interaction. These interfaces are stored in binary format and split into widget composition, configuration and behaviour. The widget's configuration includes data binding directly from the database, and the behaviour can execute arbitrary code from any module.

Our target is Angular and Typescript, which implies: (i) the usage of Angular HTML for the composition and part of the widget's configuration, (ii) the usage of CSS for part of the aesthetics configuration, (iii) the usage of Typescript for the implementation of the widget's behaviour.

- Requirement: To migrate the GUI visual and behavioural aspects [Verhaeghe 2021a, Garcés 2017, Moore 1994, Moore 1996].
- Challenge: Model the visual and behavioural aspects of a GUI.
- Challenge: Transform these models into front-end and back-end target systems.

3.1.7 Developer's expertise related aspects

The outcome of an industrial software migration is a target system that human beings must maintain. The company aims to have a team to maintain the target system composed of members of the source system and experts on the target technologies.

- Requirement: As much as possible (considering the architectural changes), artefacts (classes, functions, forms) should be migrated in a way that reduces the effort of developers to recognise them afterwards and know where to find them [Verhaeghe 2021a].
- Challenge: The loss of information in the migration must be reduced as much as possible.

A second traversal aspect related to maintainability is readability.

- Requirement: A target technology developer must be able to read the migrated target system and, as much as possible, use the target technology etiquette. [Verhaeghe 2021a].
- Challenge: We must use normal target concepts and etiquette while not respecting the previous requirement.

3.1.8 Summary

A (semi-) automatic approach to migrate Microsoft Access to web technologies includes multiple requirements related to detailed aspects.

Agnostic: The approach must be usable on multiple technological targets.

Programming language migration:

Transform. The approach must translate and transform code to the expected targets: Java, Typescript and HTML.

Import. The approach must produce a target including explicit dependencies declaration (by using import).

Library and infrastructure migration:

Replace Libraries. The approach must produce target systems using target libraries: For example, the “long” type usages in Microsoft Access must be migrated as “BigInteger” in Java or “bigint” in Typescript. MsgBox function usages are transformed using “alert” in Typescript or a Log4j “logger” in Java with SpringBoot.

Replace infrastructure. The approach must produce target systems providing similar infrastructure by using target libraries and frameworks.

Paradigm migration:

Procedural. The approach must migrate procedural code in terms of classes, methods and objects. Note that we do not pay close attention to the automatic solution of this requirement in the thesis.

Access First-Class Citizens. The approach must produce migrate MS Access First-Class Citizens in terms of classes, methods and objects.

Architectural migration:

Partial. The approach must migrate only the pieces of software needed on the target.

Split. The approach must split the source system into front-end and many possible back-end targets.

Communication. The approach must establish communication between the different target artefacts.

GUI migration:

Visual. The approach must migrate the GUI visual aspect.

Behaviour. The approach must migrate the GUI behavioural aspect.

Developer:

Expertise. The approach must, as much as possible (considering the architectural changes), migrate the artefacts (classes, functions, forms) in a way that reduces the effort of developers to recognise them afterwards and know where to find them, keeping their expertise.

Etiquette. The approach must produce a target system respecting the target technology etiquette so that a target technology developer can understand and read the migrated target system.

3.2 An industrial software migration process constraints

Due to the technical details of our case study, the holistic nature of our migration and the big picture of the modernisation project at large, we inherit multiple requirements. Our requirements aim for the tooling of migration processes. The produced tools must take into account, therefore, the process constraints.

Agile, Iterative and Incremental:

Plan oriented: No approach can replace a plan, especially with the size of applications and the complexity of our migration. The approach must reduce the restrictions to respond to the migration plan instead of the migration plan responding to the approach.

Core features first: Along with the logic of Agile development, our target migrated systems should provide the most important features first, reducing the need to migrate code irrelevant to the feature.

Arbitrary-size tasks: Regardless of the source code size, the nature of migration could be extremely hard or easy, depending on the migration target. We should be able to migrate random size pieces of software: from packages and classes to specific attributes, methods, etc.

Immediate feedback: Feedback is chief in our approach. We need the different stakeholders to be able to measure the impact of their decisions.

Right to be wrong: Favouring experimentation is important to learn the decisions to make in a migration. As feedback is important to learn the impact of a decision, reducing the impact of changing your mind is also important.

Independent lifecycles:

Permanently evolving source: The source applications will evolve by adding new features and fixing bugs. The approach must take into account the mutations.

Permanently evolving target: The migrated target applications must be shipped as soon as possible, regardless of whether the migration is finished. Once shipped the first time, it will need to evolve by adding new features and fixing bugs.

3.3 Shortcomings of software migration: Why a holistic approach?

In the previous chapter, we aim to give a general idea of the solutions to software migration in the scientific literature. We find that each approach specialises in a specific kind of software migration. This is due to most of the cases requiring modernising a single variable. Even in the case of De Lucia *et al.* [De Lucia 2008], the variable to migrate is strictly architecture. The only case we found that systematically had to automatise migration in multiple levels is De Souza *et al.* [de Souza 2003].

In the grey literature, we find Brodie and Stonebraker [Brodie 1995], who propose an iterative, incremental methodology to software migration: Chicken Little, this approach is applied to multiple kinds of systems, mainly in the context of data migration, but the methodology applies to any software migration. In the same line, we find the Butterfly approach, proposed by Wu *et al.* [Wu 1997], which proposes an iterative, incremental methodology to software migration but allows the original system to remain as it is.

Regardless of this usefulness, the approach does not cover any automated tools to help along the process.

3.3.1 The holistic approach

It could be argued that a solution based on migrating one aspect at a time would be less risky and simpler. We argue that as our migration's baseline is the programming language migration, it is impossible to adequately split into multiple independent, successful migrations by kind of migration.

For example, let us try to think about splitting library migration from language migration: (i) many language features provided by Microsoft Access are supported by libraries on the target languages. (ii) We could only migrate language up to the moment when the target compiles and works if the source libraries are available in the same way on the target. This is only possible when the source and target languages share runtime or if a target library with the same API and types is available, which is not our case.

Let us consider splitting architectural migration (which implies the stripping of the application) from language migration. (First solution) To split the source application into two parts front-end and back-end applications. This would imply access to interoperability technology that we do not have in Microsoft Access (this is one of the reasons for the migration taking place). This would be possible but inadequate, as it would imply not only the implementation of the communicating technology (which should be excellent to not produce visible changes to the users) but also the splitting of the source application, with process implications such as having to delay the bugs resolution and the addition of new features to the end of the migration. (Second solution) To fully migrate the complete application to two targets and have them working before splitting would have many implications as well: We have to enable the migration of features that make no sense on target (to translate visual elements to what is meant to be the backend),

Finally, let us consider splitting the migration of the GUI from the language migration. (First solution) The GUI behaviour, like handling events or navigating from one screen to another, requires written code, often based on the translation of the source application. Therefore the GUI cannot be fully migrated without migrating the language. (Second solution) Migrating language would not be enough to produce the target GUI, as the source technology does not have source code to represent the visual aspect of the widgets but descriptive properties.

3.3.2 Shortcomings of software migration

The shortcomings of software migration literature (as a whole) are: (i) the lack of an approach working systematically with the many variables of our project and their interaction; (ii) there is no solution taking into account developers understanding of evolution; (iii) no approach articulates the different steps of the migration process: planning, understanding and transforming; (iv) very few approaches are technology agnostic and only on specific migration cases.

To illustrate these shortcomings, we provide Table 3.1, listing many approaches discussed in Chapter 2. Each column (other than the article ID) responds to one of the requirements presented in Section 3.1, and the values indicate if the article treats this requirement or not.

Table 3.1: Requirements table. This table shows multiple approaches and notes if they resolve or not each of the requirements proposed in Section 3.1. The first entry shows the information related to the approach presented in this thesis.

Requirements Article ID	Agnostic	Lanugage		Library / infrastructure		Paradigm		Architectural			GUI		Developer	
		Translate / Transform	Imports	Infrastructure	Library	Procedural	Access FCC	Partial	Split	Connection	Visual	Behavoral	Keep Names	Etiquette
1 [Verhaeghe 2019]	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
2 [Angulo 2018]	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
3 [Garcés 2017]	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✓	✓
5 [Jain 2015]	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗
7 [Ahmad 2014]	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
8 [Bergmayr 2013]	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗
12 [Brant 2010]	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗
15 [De Lucia 2008]	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗
16 [Zhou 2007]	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
17 [Di Santo 2007]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗
19 [Zhang 2004]	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
20 [de Souza 2003]	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗
21 [Martin 2002a]	✗	✓	✗	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗
22 [Zou 2001]	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗
24 [Kontogiannis 1998]	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
30 [Moore 1994]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗

3.4 Conclusion

Industrial migrations are complex and have many aspects to consider. From the decadence of the code, which threatens any migration by an excess of accidental complexity; to the obsolescence of the system, which impacts what is possible, also threatening any migration by technological limitations.

A lot of work has been done to approach different aspects of a software migration, but not much in systematising their application of them into a single approach. In the next chapter, Chapter 4, we propose the state of the art about different technologies required to develop a new approach. These technologies have shortcomings because of the challenges and constraints discussed in this chapter.

CHAPTER 4

State of the Art: Modelling, Understanding and Transforming

Contents

4.1 Software migration as iterative reengineering 36
4.2 Modelling migrating applications 37
4.3 Understanding migrating applications 42
4.4 Transforming migrating applications 44
4.5 Conclusion 48

As our approach is based on multiple artefacts of different kinds, in this chapter, we analyse the state of the art of reengineering processes for migration, modelling, understanding and transforming.

In Section 4.1, we present the main reengineering processes used in software migration. In Section 4.2, we analyse the modelling options, particularly software migration modelling, passing by intermediate languages, structures and how to reverse engineering programs to produce them. In Section 4.3, we analyse options to understand and measure complexity from the point of view of a real industrial case of software migration. In Section 4.4, we study different approaches to transforming software aiming for software migration. Each section ends with the shortcomings of the existing solutions.

The chapter ends with Section 4.5 with the requirements for a successful approach.

4.1 Software migration as iterative reengineering

Software migration is firstly a process, as we already discussed in Section 2.6. In this section, we discuss the processes taking reengineering as a whole.

Brody *et al.* [Brodie 1995] Migrating Legacy Systems.

The Chicken Little Strategy gradually rebuilds the legacy system on the target platform using modern tools and technology. The legacy and target systems make up a hybrid system communicating the legacy and target systems during the reengineering process. As part of the process, analysis and decomposition of the legacy system, reengineering, and migration of interfaces, applications, and databases are required.

Wu *et al.* [Wu 1997] The Butterfly Methodology: A Gateway-Free Approach for Migrating Legacy Information System.

The Butterfly methodology focuses on legacy data migration and develops the target system as a separate process. As a feature of this methodology, the legacy and the target data system can continue to operate.

Bianchi *et al.* [Bianchi 2003] Iterative reengineering of Legacy Systems.

The Iterative Reengineering methodology recovers incremental aspects from The Chicken Little approach, such as decomposition and analysis of the legacy system. It recovers from the Butterfly approach, the feature of allowing both systems to coexist.

These approaches are focused on data storage and representation. We do not address this matter, as we expect to use the same database. We remark from these approaches that the main driving force is the process: planning, iterativity and incrementality.

We also based our approach on the following two articles:

Khadka *et al.* [Khadka 2014] How do professionals perceive legacy systems and software modernisation?

It proposes a study of what stakeholders of a software migration expect of such a project. This article taught us that we need to tool the whole process, as there is a lack of knowledge and ownership of the migrated system; the process is error-prone, the resources of the system modernisation compete with those of the software maintenance, the time and budget to modernise a system is always shorter than needed.

Razavian *et al.* [Razavian 2012] A Lean and Mean Strategy for Migration to Services. It proposes a study of what experts usually do in migration to service

architecture projects. This article taught us that experts do not rely on reengineering tools. This has a straight implication: human force is enough for architectural migration when the planning and approach are sound. To not get in the way of the planning and the approach, tools must be oriented to specific steps of a process and be optional.

4.2 Modelling migrating applications

We use models to think and understand all problems [Ludewig 2003]. We turn to intermediate representations in software migration, especially when migrating language. Intermediate representations are models used to represent source code within different tools with different goals [Kienle 2010]. Virtual machines and compilers use intermediate representations to transform computational models into real behaviour, partially abstracting the target hardware and operative platforms (a similar kind of detachment is required when migrating).

The primary domain of intermediate representations is compilers and virtual machines, where the code requires a computational model designed to enable optimisation and translation to machine code. As language migration requires expressing a source's perceived behaviour into a target technology, it is required to understand how the implementors of behaviour use these models.

Because of their accuracy, these models are popular in software analysis, reverse engineering, and software modernisation. Accuracy is a critical feature of Intermediate representation (IR). An IR should never lose important information from the source code [Chisnall 2013].

There are two kinds of representation: languages and data structures.

4.2.1 Intermediate languages

The C language. The C language is a low-level language allowing to do most of the things that assembly can offer, giving a thin layer of hardware architecture abstraction and counting with several powerful compilers in multiple platforms and operative systems. Like this, the usage of this language became a way to express equivalent behaviour in multiple targets. C is used by many language implementations such as Eiffel, Haskell (Glasgow Haskell Compiler), Gambit Lisp, Squeak and Pharo's Smalltalk subset Slang, Cython, etc.

Intermediate language by design. Intermediate languages provide a layer of abstraction from the environment where the expressed code is used. The most prosperous domain of usage is virtual machines and portable code (p-code) machines. Intermediate languages are proposed to express computations that can be optimised and adapted to a specific operative system and hardware architecture at execution.

Some examples of these intermediate languages are the bytecode in Pascal [Overgaard 1980], Smalltalk, Pharo[1], Java[2]. The Parrot intermediate representation[3], the Three Address Code[4], Register transfer language [Davidson 1980], GCC's GENERIC and GIMPLE (two intermediate representations with different degree of abstraction) [Merrill 2003], LLVM intermediate representation [Lattner 2004][5], Microsoft's Common Intermediate Representation[6] for .NET platform[7].

4.2.2 Intermediate internal structures

Intermediate languages rely on data structures to represent the expressed information. Different structures meet different requirements [Kienle 2010]. In this Section, we only cover structures related to those we use. We, therefore, dismiss other structures such as Control Flow Graphs, Dependency Graphs, Call Graphs, and the multiple uses of Directed Acyclic Graphs.

4.2.2.1 Source code models

Here we present those structures that lose only information that can be reproduced, such as grammatical specificities.

Abstract Syntax Trees (AST). An AST is a tree structure representing the structure and content of a program, disregarding the specifics of the grammar. Introduced by Algol 60 [Wirth 1966] and popularised by Aho *et al.* [Aho 1972, Aho 1986]. AST is recurrently used in source code translation [Brant 2010, Kontogiannis 1998], software analysis [Ira Baxter 1997, Baxter 1998], and for building other models [Ducasse 1999b].

Abstract Semantic Graph (ASG). A popular graph version of an AST is the Abstract Semantic Graph (ASG). An ASG is an Abstract Syntax Tree extended by linking uses with definitions or declarations, making it into a graph.

eCST. Rakic *et al.* [Rakić 2015] proposes eCST. This model is an angular stone to provide multiple analysis features based on a single model, which unifies multiple ASTs.

[1] https://scg.unibe.ch/download/st/11Bytecode.pdf
[2] https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions
[3] http://docs.parrot.org/
[4] https://en.wikipedia.org/wiki/Three-address_code
[5] https://llvm.org/docs/LangRef.html
[6] http://www.ecma-international.org/publications-and-standards/standards/ecma-335/
[7] http://learn.microsoft.com/en-us/dotnet/standard/managed-code

Abstract Syntax Tree Meta-model. OMG: Object Management Group[8] proposed architectural modernisation meta-models[9]. They offer standard meta-models based on similar ideas to eCST Perez-Castillo *et al.* [Pérez-Castillo 2011]. ASTM: Abstract Syntax Tree Meta-model, which uses imaginary nodes in enriching AST, representing the program below the function level. As ASTM homogenously unifies concepts, it loses some information that cannot be recovered.

4.2.2.2 Knowledge models

Here we present those structures that extract higher-level knowledge but, at the same time, lose information that cannot be reproduced.

Unified Modelling Language (UML). UML has been proposed by OMG [Software 1997]. It consists of a unified graphical language helping to visualise the design and structure of complex software systems. UML is a meta-model providing specific kinds of entities that are common to object-oriented languages. Along with UML, we often find Meta Object Facility (MOF) proposed by OMG [Group 1997]. MOF is a meta meta-model which is used to define UML.

Knowledge Discovery Meta-model. KDM: Knowledge Discovery Meta-model[10], which represents a higher level of knowledge, aims to describe an application's requirements. Along with the same line, Kent *et al.* [Kent 2002] proposes Model Driven Engineering, which uses models built out of meta-meta-modelling to represent different features and presents model transformation as a higher-level operation for modifying a system.

Famix. Ducasse *et al.* [Ducasse 1999a], Demeyer *et al.* [Demeyer 2001] and Anquetil *et al.* [Anquetil 2020] present Famix and newer versions, which are Meta-Meta-Model facilities representing analytic models and used by the Moose platform[11], which based on this description of the meta-models can produce reusable tools. FAMIX models the different structural and behavioural entities of a program, keeping association information (see Figure 5.2).

Verhaeghe *et al.* [Verhaeghe 2021b] proposes Casino, which uses Famix to represent a visual model for extracting GUI knowledge and use it as a pivot model to migrate visual requirements agnostic from source and target technology. We use this meta-model in our approach to migrate visual aspects.

Zaragoza *et al.* [Zaragoza 2022] uses Famix to represent architectural models to propose multiple micro-services candidates.

[8] Object Management Group(OMG), http://www.omg.org/
[9] Architecture-driven modernisation, http://adm.omg.org/
[10] Proposed by OMG
[11] https://modularmoose.org/

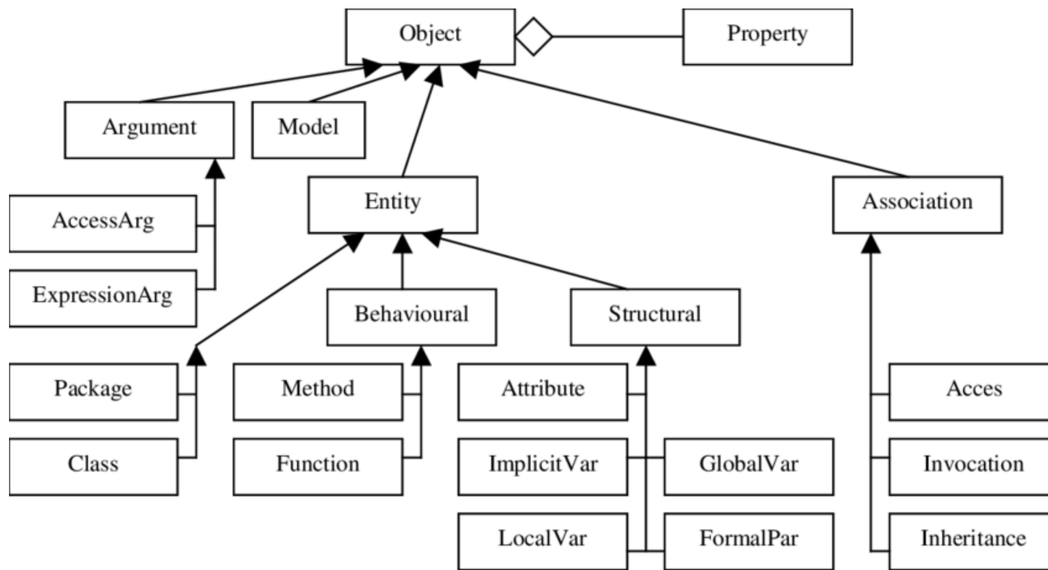


Figure 4.1: Famix meta-model as described by Demeyer *et al.* [Demeyer 1999b]

Migrating models. The models we have found in the literature are great for software analysis and compiling or interpreting code. However, these models do not provide insight into software migration. The representations discussed first are highly specialised for a specific language's compiler or virtual machine. AST is often specialised in the language they represent. eCST and ASTM put together different AST nodes using imaginary nodes, which do not unify much of the structure. ASTM aim to unify concepts by losing details. Famix and UML/MOF models need more information to be used in language migration. Meta-meta-model-based approaches produce either a meta-model per technology (Famix) or one meta-model that homogeneously represents multiple languages (UML). Using one separate meta-model per language punishes the migration of entities heavily since we need to provide transformations for any element, even if the element to transform is identical in source and target technologies, *e.g.* if different models represent source and target, to migrate an IF statement requires explicit mapping. A single meta-model representing paradigm concepts cannot represent each language's semantic and grammatical richness, as it unifies versions of the same concept losing critical details in a migration.

4.2.3 Reverse-engineering code

Extracting a model. Models are as crucial as the way to extract them. When classifying models by means used for extraction, we find three families: static analysis, dynamic analysis and reflective API. The different means used for extraction constrain the models differently and enable or disable them to fit different kinds of use.

Static analysis: It leverages compiling and interpreting techniques, such as parsing, over plain text or binary files. The outcome allows the users to access both program and configuration representations. [Artho 2001, Bacon 1996, Cloutier 2016, Leroy 2003, Nagappan 2005, Rakić 2015].

Dynamic analysis: It instruments applications and produces various traces, which can be used to understand different aspects of a program. [Ball 1999, Cornelissen 2008, Ducasse 2004, Greevy 2005a, Kuhn 2006, De Pauw 2002].

Reflective API: It builds models by querying a reflective system through interoperation. We did not find articles on this way of conducting model building, but we found software doing it: Famix loads Squeak and Pharo models by interacting with the reflective API of the system[12].

Hybrid approaches: Those approaches mix static and dynamic analysis [Francesca 2008, Gotti 2016, Gustafsson 2000, Richner 1999, Silva 2013, Stroulia 2002],

4.2.3.1 Extracting models for migration

In software migration, particularly in language migration, static analysis is compulsory: the expected outcome is a system readable by humans with a similar structure to the source application (see Section 3.1.7). However, textual representation for running static analysis is not always available. Some 4GL only have binary/proprietary representations. Some workarounds are proposed in these cases to obtain some extracted structure textual representation.

Garcés *et al.* [Garcés 2017]: This article proposes to parse all the files representing Oracle Forms to obtain a model. The article does not specify the kind of file they used for analysing. But according to [Sánchez Ramán 2010] and the existence of Oracle exporting tools[13] from Form to XML, we suspect that they follow the same path.

Sánchez *et al.* [Sánchez Ramán 2010]: This article proposes to reverse engineer GUI Layouts from Oracle Forms. They export the Form structure as XML and use EMF[14] tools for generating models.

Shah *et al.* [Shah 2011]: This article points out the complexity of accurate GUI analysis by code interpretation. Extracts a technology-agnostic UI model by crawling the application run-time using AOP, enabling android portability.

[12]https://github.com/moosetechnology/moose
[13]https://blogs.oracle.com/apex/forms-to-apex-converting-fmbs-to-xml
[14]http://www.eclipse.org/modeling/emf/.

To the best of our knowledge, in the context of software migration, most of the approaches use static analysis: (1) the primary technique is parsing: either the source code of the source application or the exportation from a binary format to a text format (for example, XML/HTML). (2) The approaches are based on batch processing instead of online access. Some of these model-extracting tools VerveineJ[15] extracting Famix models out of Java projects, Proleap[16] extracting ASG out Visual Basic 6 projects, SmaCC[17] and ANTLR[18], providing parsers to extract AST from multiple languages.

4.2.4 Reverse engineering and modelling approach shortcomings

Reverse engineering. The means to extract models out of 4GL languages, besides leveraging specific features of the source application technology (oracle forms extractors), do not extract source code. Not having access to the source code makes it difficult to conduct a language migration. In Chapter 5, we return to this problem.

Modelling. AST and ASG structures lose no relevant information. Still, both are tangled with the technology they represent, threatening the ability to represent multiple languages with the objective of language migration. eCST uses intermediate nodes to unify the knowledge within a concrete syntax tree; it loses no relevant information and includes even grammatical details as separators. eCST uses the same nodes for many concepts shared between different languages. However, it lacks the linking information provided by an ASG. as the underlying tree is tangled with the technology, ASTM unifies concepts, losing information that is essential to language migration.

KDM and Famix models lose too much information required to conduct any migration that entails the migration of behaviour, such as language or library migration.

These modelling approaches either (i) aim to have a model per language to ensure correctness, increasing the complexity of the implementation of model transformations, or (ii) aim to unify concepts, losing information specific to each language which threatens the language migration. In Chapter 6, we return to this problem.

[15]https://github.com/moosetechnology/VerveineJ
[16]https://github.com/uwol/proleap-vb6-parser
[17]https://refactory.com/smacc/
[18]https://www.antlr.org/

4.3 Understanding migrating applications

Often when understanding the reality of our software, as we do in any other science in their domains, we turn to means to simplify the excessive complexity of a real-world system into something more accessible to grasp: Metrics and visualisations.

4.3.1 Metrics

Metrics allow understanding specific qualities of a piece of software from the point of view of maintainability and understandability [ISO 2011a, ISO 2001]: how hard it is to evolve and modify an application and how easy it is to understand it.

Measuring complexity. There are many complexity metrics for software. The most popular are Lines of Code, McCabe's Cyclomatic Complexity [McCabe 1976], and Halstead metrics [Halstead 1977]. All of them aim to measure the complexity of a piece of code by its construction.

Measuring object-oriented design quality. Basili *et al.* [Basili 1995] validates multiple metrics aiming to measure the quality and maintainability of Object-Oriented systems, with metrics such as Weighted Methods per Class, Depth of Inheritance Tree of a class, Number Of Children of a Class, Coupling Between Object classes, etc. Ducasse *et al.* [Ducasse 2001b]

Measuring architectural quality. Emerson *et al.* [Emerson 1984], Bieman *et al.* [Bieman 1994] and many others [Abdeen 2011, Anquetil 2013, Allen 2001, D'Ambros 2009, Aruna 2008, Balmas 2009, Bansiya 1999, Binkley 1998, Blondeau 2015, Briand 1997, Demeyer 1999a, J. Eder 1994, Lakhotia 1993, Lee 1995] propose different ways to measure cohesion and coupling at different levels: modules, classes, artefacts, and to understand the impact of these measures with the quality of software and maintainability.

Measuring reengineering and migration cost. Sneed *et al.* [Sneed 1999] proposed a method to ponderate the risks of reengineering tasks as a relation between the maximum risk and the cost of the task. Afterwards, Sneed *et al.* [Sneed 2004] present a cost model for software maintenance analysing different tasks and proposing a ponderation system. Finally, Sneed *et al.* [Sneed 2020] present a Cost-driven migration.

Basant *et al.* [Namdeo 2022] propose a cost and risk model for database migration in the case of migrating from relational databases to NoSQL.

Regardless of the history of software migration and metrics, we found no systematic measuring of the complexity of a Software Migration. In contrast, cohesion

and coupling are used for proposing automatic proposals for migrating towards Object Oriented models, Services, Micro-services,...; no metric measures the code that would be considered an architectural violation on the target, nor the technological distances between a source and target technologies.

4.3.2 Visualisations

Visualisations allow us to quickly understand specific properties of the structure of a system or parts of a system.

Many approaches, mainly based on visualisation techniques [Healey 1992,Tufte 1997, Tufte 2001,Ware 2000], have bloomed to represent the software structure [Ducasse 2005, Dong 2007,Langelier 2005,Lanza 2003,Lungu 2006,Wysseier 2005,Wettel 2007] Visualisations are often related to Software Understanding for maintenance. We focus principally on structural and dependency visualisations.

System level: Langelier *et al.* [Langelier 2005] propose a visualisation framework that supports quality analysis of large-scale software systems. Lanza *et al.* [Lanza 2003] worked on system-level understanding combining metrics and visualisations. Lanza *et al.* [Lanza 2001] and D'Ambros *et al.* [D'Ambros 2006, D'Ambros 2007] worked on understanding evolution and finding bugs. Govin *et al.* [Govin 2017, Govin 2016] works on visualising architecture based on artefact tagging for assessing architectural recovery.

3D system level: Feijs *et al.* [Feijs 1998] proposes an architectural visualisation to understand the interconnection between different artefacts. Wettel *et al.* [Wettel 2008,Wettel 2007] proposed code city as a way to understand system code as the distribution of a city. Greevy *et al.* [Greevy 2005b] exploit a third dimension to visually represent the dynamic information, namely object instantiations and message send.

Package level: Ducasse *et al.* [Ducasse 2006] show how properties are spread in a population of packages. Laval *et al.* [Laval 2011] worked on the understanding of package evolution.

Class level: Those visualisations producing a class (as in object-oriented) overview. Ducasse *et al.* [Ducasse 2005] and then Agouf *et al.* [Agouf 2022] worked on versions of the so-called class blueprint, which mixes metrics and visualisations at the level of a class.

There are many architectural visualisations out there. Still, these visualisations are there to spot what is reality and interconnection. We found no visualisation relating the classes with the architecture. We did not find any architectural overview that allowed us to grasp the architectural complexity. We understand *complexity* as the tangling of the different architectural concerns, expected to be split in the target technologies.

4.3.3 Metrics and visualisations shortcoming

There is a great deal of work done on metrics, visualisations and mash-ups of metrics and visualisations. However, during this *industrial* thesis project, we faced the need to measure and estimate software migration efforts to understand risks and costs. We found no metrics or visualisation covering the technological and architectural gap between the source and target technologies. We return to this problem in Chapter 7.

4.4 Transforming migrating applications

White-box and grey-box approaches require the application of transformations. Specific transformations could automate black-box approaches such as wrapping (for approaches, see Section 2.5). To understand the existing literature and shortcoming of approaches on transformation, we analyse two main domains: Program Transformation and Model-to-Model Transformation.

This section does not consider *refactors* [Balaban 2005, Bart Du Bois 2006, Brant 1998,Dig 2005,Ducasse 2001a,Fowler 1999,Murphy-Hill 2009,Opdyke 1992, Pérez-Castillo 2014, Porres 2003], as they provide transformations with specific properties (such as not changing behaviour) within the same program.

4.4.1 Program Transformation

A program transformation is any operation that takes a program and generates another program. The logic of this transformation for effective program transformation leverages language semantics and grammatical knowledge. [Appel 2002, Aho 1986,Ducasse 2022,Kontogiannis 1998,Martin 2002a,Moynihan 1991,Partsh 1983, Plaisted 2013,Plaisted 2021,Sakamoto 2013,Visser 2004,Visser 2005,Wirth 1966]

There are multiple surveys of strategies in program transformation systems [Partsh 1983, Visser 2005], with different approaches. Partsh *et al.* [Partsh 1983] survey program transformation as mechanisms to aid programming evolution; Visser *et al.* [Visser 2005] surveys rule-based systems and reusability to cope with software evolution problematics.

We are particularly interested in this domain of study for translation and rewriting (also known as *rephrasing*, often implemented by *tree rewriting* techniques) applied to software migration for language and library migration. Language migration has been addressed successfully by language translation techniques [Brant 2010, Kontogiannis 1998, Martin 2002a, Sakamoto 2013]. Also, library migration has been addressed successfully by program transformation techniques [Ducasse 2022].

However, the main drawbacks of Program Transformation are: (1) the small or nonexistent reusability across different languages and the fact that most decisions require a deep understanding of the different language technologies, and (2) the

transformation done at the level of functions or statements is too shallow to enable architectural, infrastructure or paradigmatic migrations (see Section 3.1).

4.4.2 Program Trans-compilation

Program Trans-compilation is a sub-domain of program transformation that deals partially with software migration ([El-Ramly 2006, Malabarba 1999, Trudel 2012, Yasumatsu 1995]). We consider program trans-compilation to be tangential to software migration as it looks for a common objective: to execute an application in a target technology. Still, a big difference between trans-compilation and software migration, particularly source code translation and library migration, is that trans-compilation methods work for previously chosen grammatical constructions and libraries. Trans-compilers produce code that is not expected to be understood and maintained: the only important thing is that it works out of the box.

Industrial migration is expected to be maintained by teams of human beings: the result must be readable and maintainable. This makes the trans-compilations approach not suitable.

4.4.3 Model to Model Transformation

A model transformation in model-driven engineering (MDE) is an automated way of modifying and creating models. Much research has been proposed aligned with MDE [Agrawal 2003, Aranega 2014, Bergmann 2012, Esbai 2015, Etien 2015, Giese 2010, Jouault 2010, Lauder 2012, , Mellor 2002, Group 2000, Ráth 2008, Razavi 2012, Sneed 2011, Sendall 2003, W3C 1999].

Standarising. OMG [Newcomb 2005, , Group 2000] proposed some of the most popular bases for MDE for engineering and modernisation.

How to transform. The MDE community has also worked on the different ways to apply transformations to support reuse, reproducibility and incrementality for better transformation performance.

Aranega *et al.* [Aranega 2009, Aranega 2014]: This article proposes mutation analysis for model transformation based on traceability.

Etien *et al.* [Etien 2015, Etien 2010]: This article proposes the definition and execution of localised model transformations.

Kusel *et al.* [Kusel 2013]: This article surveys the incremental application of rules of model-to-model transformation, comparing Triple Graph Grammar (TGG) [Giese 2010, Lauder 2012], Atlas Transformation Language (ATL) [Jouault 2010], Viatra [Bergmann 2012, Ráth 2008] and others [Razavi 2012].

Transforming infrastructure. Along with the different studies, specific infrastructure emerged from the literature. Czarnecki *et al.* [Czarnecki 2006] provides a survey and taxonomy of the different features of a transform engine. We highlight some facts that are relevant in further chapters.

- Model transformations are implemented under the shape of some transformation engine or language that automatically applies a set of transformation rules. These rules are often composed of 2 parts: the left-hand side, node localisation and the right-hand side, node transformation.
- Transformation engines give roles to models: either source or target. There are three kinds of relations between source and target: The source and target are required to be explicit; The source and the target are the same; The transformation can create a target from a source.
- Rules are responsible for accessing the domain objects. For example, the rules access all the entities over which they should work.

Forward engineering. The literature also includes many uses, including forward engineering, as Esbai *et al.* [Esbai 2015] proposes model transformation to create Model-View-Presenter web projects.

Modernisation. Another kind of use, the one that interests us the most, is modernisation: As proposed by the MDE community, approaches of modernisation based on model-to-model (M2M) transformation establish their process and solutions on refining knowledge out of the code, abstracting it from technical details, and merging information from different sources, to be used to generate code on specific technological platforms.

Garcés *et al.* [Garcés 2017]: This article models *Oracle Forms*, extracting GUI concerns by analysing the source code and then using it to produce modernised code.

Verhaeghe *et al.* [Verhaeghe 2021b]: This article uses a model GUI concerns in a model that is used for pivoting to multiple possible destinations.

Newcomb *et al.* [Newcomb 2005] and Ulrich *et al.* [Ulrich 2010]: In the context of Architecture driven modernisation, these articles propose successive transformations over the extracted Abstract Syntax Tree model (ASTM) [Object Management Group 2011]

Sneed *et al.* [Sneed 2011]: This article propose successive model transformation applications for migrating from Cobol to Java.

The main drawbacks of the MDE approach are: (i) the lack of interactivity on the rule application. (ii) Most models in modernisation lose all behavioural information. (iii) The modelling approach of one meta-model per language punishes the rule reuse, as rules must be defined even for structures equivalent between source and target languages. (iv) The unified modelling approaches lose the semantic unicity of different languages.

4.4.4 Interactive iterative context-aware partial migration approach justification

The work in program transformations and model-to-model transformation is significant. However, as discussed in Chapter 3, we must respond to multiple constraints not supported by any of the mentioned methods:

- Empower the users to conduct the migration in a learning fashion to help them to migrate their knowledge about the system.
- To enable source and target systems to have independent project lifecycles in the context of a language migration.
- Plan-based, allowing to migrate by use-case instead of technological artefacts: to subordinate to the plan the decisions of **what**, **how**, **where** and **when** to migrate.

In Chapter 9, we come back to this problem.

4.5 Conclusion

In this chapter, we overview the state of the art of multiple domains and the limitations.

- In Section 4.1, we discussed many reengineering processes used in software migration. None of these processes is articulated with tools.
- In Section 4.2, we discussed the modelling options and how to reverse engineering programs to produce them. We found no modelling option specialised in software migration. We found no Microsoft Access reverse engineering approach.
- In Section 4.3, we discussed options to understand and measure the complexity of software migration. We found no metrics measuring the technological gap to cover during a software migration. We found no architectural visualisations allow us to understand the system's architectural complexity or the architectural implications of a class.

- In Section 4.4, we discussed different approaches to transforming software aiming for software migration. We found no approach resolving systematically the different parts of this industrial migration (see Chapter 3). We found no approach that subordinates the major decisions (**what**, **how**, **where** and **when** to migrate) to the plan. No approach responds to the process and migration requirements discussed in Chapter 3.

Part I

Reverse engineering and Modelling

CHAPTER 5

Reverse engineering a Microsoft Access project

Contents

5.1 Enabling reverse engineering for Microsoft Access 52
5.2 Microsoft Access: a partially observable system 52
5.3 Component Object Model (COM) technological overview 55
5.4 Mixing static and internal access information 58
5.5 Validation 60
5.6 Threats to validity 68
5.7 Discussion 69
5.8 Conclusion 70

This thesis focuses on the modernisation of Microsoft Access systems. As explained in Chapter 4, attempts to produce (semi-) automated solutions to software migration requires models extracted through reverse engineering. Like many other Rapid Application Development (RAD) – Oracle Forms, Flash, Flex, etc. –, Microsoft Access uses a proprietary binary format for storing the programs.

This chapter presents three research questions to guide the study (Section 5.1). Section 5.2 presents Microsoft Access as a *partially observable* domain and stress its *opacity*, and how these limitations threatens reverse engineering. Section 5.3 overviews the COM interface to Microsoft Access and summarises the information that can be accessed and the challenges of putting it together into a model. Section 5.4 presents an approach mixing static and internal Access information. Section 5.5 validates the approach to be used in Software Migration by measuring the information loss of the produced model over ten projects, eight of them industrial. Section 5.6 lists the threats to the validity of our validation. Section 5.7 discusses the validation results, adding insight into the outcome of the validating process.

We note that the content of this chapter is based on our article “Analysing Microsoft Access Projects: Building a model in a Partially Observable Domain” Bragagnolo *et al.* [Bragagnolo 2020a].

5.1 Enabling reverse engineering for Microsoft Access

Most programming language compilers use plain text files as input for the programs they should compile or for configuration, such as XML, YML, properties, etc. Therefore, reengineering tools [Kienle 2010] often use the same approach for producing their internal models [Fleurey 2007, Garcés 2017, Hayakawa 2012, Newcomb 2005, Sánchez Ramán 2010, Ulrich 2010, Verhaeghe 2019]. However, not all languages are based on text files. Some use binary formats, such as Microsoft Access, Oracle Forms, Flash, Flex and many other Rapid Application Development (RAD). In our particular case, we study the applications developed in Microsoft Access. Microsoft Access uses a proprietary binary format for storing the programs. Due to this policy and the lack of exporting capabilities, a Microsoft Access application lacks full-text representation.

As we specify in Chapter 3, our migration involves (i) language migration, (ii) architectural migration and, (iii) UI migration. Taking into account that we need to be able to conduct all these migrations with their implications, we propose the next three research questions to lead the research:

#RQ1: Can we obtain an application representation by querying the IDE runtime?

#RQ2: Are we able to re-engineer the meta-data sources into a model for migration?

#RQ3: Does the obtained model provide enough accurate information for conducting a language migration?

Using these questions as general guidelines, we propose a model built on binary sources by applying reverse engineering on the run-time of the Microsoft Access development environment and re-engineering to transform the available data into a unified model.

In Section 5.3, we address RQ1 by overviewing and analysing the possible interoperations with the IDE. In Section 5.4, we address RQ2 by analysing the challenges of transforming the information provided by the IDE into a model and a solution. Finally, in Section 5.5, we address RQ3 by measuring the loss of information when conducting a simple software migration using the proposed model.

5.2 Microsoft Access: a partially observable system

Microsoft Access is a relational database management system (RDBMS) that offers a graphical user interface and software-development tools. We briefly present and stress the critical problems in extracting information about Access applications.

5.2.1 Microsoft Access

Visual Basic for Applications (VBA) is provided as a programming language. VBA is an object-based extension of Visual Basic [Wegner 1987]. Microsoft Access is a fourth-generation language (4GL), comparable with Oracle forms or Visual Foxpro. It has the same mission of easing GUI creation.

Microsoft Access proposes a hybrid paradigm that aims to tackle down GUI, data storing and processing in a fully controlled and centralised environment. A program developed in Microsoft Access orchestrates its first-class citizens: forms, modules, classes, tables, queries, reports and macros.

To ease the work of GUI development, Microsoft Access provides a point-and-click GUI Builder. Like many other GUI builders in the market, such as Android Studio, Eclipse or Microsoft Visual Studio, Microsoft Access also has to face the problem of distinguishing the generated content from the hand-crafted content. Android Studio uses the R class [1] for scoping generated code, Visual Studio.Net uses partial classes [2], and JavaFX uses XML files and annotations.

In the case of Microsoft Access, Forms and Reports are split into two parts: (1) the VBA code, produced and modified only by the developer, (2) the internal component structure, produced and managed by the IDE, accessible to the developer only through point and click.

Microsoft Access uses a proprietary binary format. This format organisation is undocumented, implying that extracting data directly from the file would require a substantial reverse-engineering effort. Furthermore, Microsoft Access uses entity-specific formats for each first-class citizen type. In some cases, such as forms and reports, it has two formats to respond to the internal division required for managing code generation. This variety of formats leads to a more complex problem, threatening the generalisation of a solution.

5.2.2 Limited exporting

Microsoft Access provides a visual interface to export some entities by point and click. This process is time-consuming and prone to error. It is not tractable for complete applications; not all the elements can be exported. This leads to what we call a *partially observable* domain since we cannot obtain artefact to analyse using this tooling.

Figure 5.2 shows a simplified model of Microsoft Access's main elements. In grey, we show the elements that **cannot** be exported from the GUI, and in white, those that can. Most structural entities are not available for export, such as table definitions, SQL query definitions, reports and forms structures, and macros. The

[1] https://developer.android.com/reference/android/R

[2] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods

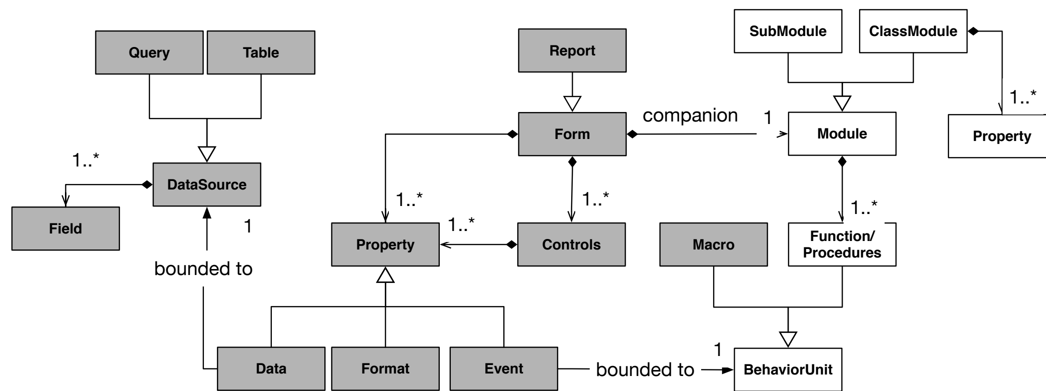


Figure 5.1: Access simplified model

main GUI exporting features are related to the Visual Basic part of the project, including modules, class modules, and the report or form companion modules. The latter is useless since their structure and configurations are not exported. By structure, we mean the attributes and their types. By configuration, we mean all the default values and descriptions that define the final behaviour of the widgets. The result is a piece of code that defines functions and procedures that access variables of unknown type that the system may use or not according to an unknown configuration, with unknown data also due to the configuration.

5.2.3 Programmatic exporting

Microsoft Access provides a binary API for inter-process communication named Component Object Model (COM). This API provides an undocumented function for programmatically exporting a text representation for all first-class citizens. Using this function requires the implementation of specific software. This function is leveraged by third-party vendors that propose enhanced exporting features for control source version purposes. Solutions such as Oasis[3], Ivercy[4], and others. We compare and extend on this subject in Section 5.7

5.2.4 Requirements & constraints

Our work happens in the context of industrial research on migration from Microsoft Access to different kinds of technologies.

We aim to migrate complete applications from one monolith origin to a front-end and a microservice-based back-end.

The main features expected for the migration process are (1) selectiveness (the developer must be able to choose what he wants to migrate). (2) iterative (propos-

[3]2023 https://dev2dev.de/index.php?lang=en
[4]2023 https://ivercy.com/

ing short loops of migrations that are easy to verify and cheap to reject). (3) interactiveness (establishing a dialogue with the developer to achieve the selective migration better).

Such a migration process must coexist with an original project under maintenance and development. These requirements imply the following constraints: (1) The CPU and memory usage must be scoped to selective migrations (the migration solution must be able to run cheaply in the working environment of the developers without performance penalties). (2) The model must be able to supply as much data as possible. (3) The model must be able to supply up-to-date data (all modifications should be reflected immediately in the model).

Following the direction of [Sánchez Ramán 2010,Garcés 2017] that works over the model of Oracle Forms, we recognise the importance of having a model based on the first-class citizens of the language. Following the abstract idea behind [Object Management Group 2011], we propose a representation close to the language that responds to Figure 5.2.

5.3 Component Object Model (COM) technological overview

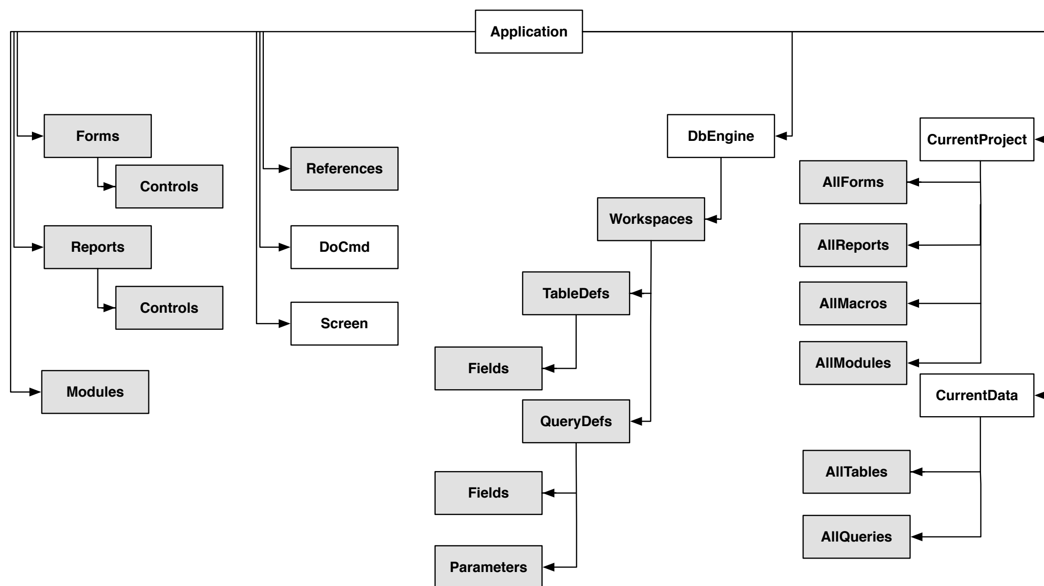


Figure 5.2: Access COM Object model. In white, those elements we can use directly. In grey, those elements depend on other elements and specific interactions.

Microsoft offers COM as an API for inter-process communication. It is present in many of their products. In this section, we provide a technological overview

that will be used to enumerate the challenges of a solution based on the usage of Microsoft Access COM API.

5.3.1 Microsoft COM & Access

Through COM, Microsoft Access exposes a reflective API that allows interoperability between different applications.

Microsoft Access documentation [5] is heterogeneous. For some uses, it provides detailed content with examples. For other uses, it does not include more than one line explaining the title. This lack of information impacts the ability to do reverse engineering, increasing complexity.

We insist on a technological overview to help answer our *#RQ1: Can we obtain an application representation by querying the IDE runtime?* and shed light on the challenges.

5.3.2 Data access

For interacting with Microsoft Access through COM, we must interact with an object model. This object model is obtained by executing COM primitives and accessing memory addresses that belong to the Microsoft Access running instance; we call these remote memory addresses remote handles.

Figure 5.2 shows the different parts of the Microsoft Access COM. We explain each part below.

Application. We get access to the application by requesting an instance of the Microsoft Ole32 library, which returns a remote handle to an application object. This application object is bound to a running instance of Microsoft Access. It exposes an explorable API and allows access to the project components directly or indirectly.

DoCmd. (Do Command) is an object that reifies most of the available operations to apply on the application. It must be used for opening a project, databases or others. Most of the objects below have this object as a dependency.

References. This collection contains *Reference objects* describing a project's static dependencies.

CurrentProject. Depends on *DoCmd*. It holds basic metadata for each element in the project by pointing to the collections *AllForms, AllReports, AllMacros, AllModules* that contains objects describing each form, report, macro and module correspondingly.

[5] https://docs.microsoft.com/en-us/office/vba/api/overview/access

CurrentData. Depends on *DoCmd*. It holds metadata for each element related to data structures. In this object, the available collections are *AllTables, AllQueries* that contains objects describing each table and query correspondingly.

DbEngine. Depends on *DoCmd*. It is the main access point to the data model. It provides access to *workspaces*.

Workspace. Depends on *DbEngine*. Represent database schemes and provides access to the scheme elements by pointing to the collections *QueryDefs* and *TableDefs*.

TableDef and QueryDef. Depends on *Workspace*. Each of these objects contains a description. For the TableDefs name and *fields*. For the QueryDefs name and SQL.

Forms, Reports and Modules. Depend on *DoCmd*. We have three main collections where we can find the Form, Report and Module objects with their inner composition. This internal definition includes composed controls (textbox, labels, etc.), properties (layout, naming, companion module, etc.) and VBA source code.

5.3.3 COM model re-engineer challenges

Re-engineer COM data into a unified model faces the following challenges:

The reading of a property of a COM entity may give back another COM entity. In some cases, we are going to read native-type or self-contained information. But in other cases, an attribute's value may be another handle. We must map the read value with a model entity type for these cases. To do so, we need to know the type of the received entity in advance.

One model entity may correspond to more than one COM entity. The COM model provides two different objects for representing each first-class citizen. For example, COM object *AllForms* contains the form's metadata, and COM object *Forms* contains a form internal representation.

One COM entity type may be mapped to different model entity types. Most of the objects in the COM model have properties represented with the same type, But to structure the analysis (visiting, for example), we need them to belong to different classes. This implies that some entities with the same type may have to be mapped to different types in our model.

Loadable objects. Many Microsoft Access first-class citizens must be dynamically loaded as COM objects to reach their internal information. For loading, they require access to many COM entry points. This implies that some objects require specific extra steps to acquire the desired data.

Summary. The overview shows that COM delivers extensive access to the binary model of a Microsoft Access application. This answers to *#RQ1: Can we obtain an application representation by querying the IDE runtime?*. This remote binary representation (from now on *COM model*) is also a very low-level model that responds to the need for interaction between applications. We also understand that an approach in this direction must respond at least to the traditional challenges of reverse and re-engineering processes.

5.4 Mixing static and internal access information

To answer *#RQ2: Are we able to re-engineer the meta-data sources into a valuable model for migration?*, we propose the approach of an online model from the point of view of migration, followed by an implementation of the proposed approach and an explanation of how our approach and implementation address each constraint and challenges stated above.

5.4.1 Approach

As a general approach, we propose to define our model as an online projection of the COM model. By **online**, we mean that the COM bridge obtains all the data on demand. This way, any change in the code immediately impacts our model. To achieve this, let our model use COM as a back-end. Our model must conform to the meta-model proposed in Figure 5.2. By delegating to COM, we aim to gather all the possible data from the analysed software on demand. We expect this strategy to give immediate feedback and allow quick and agile modifications over the information without requiring further extractions, reducing the need to plan the data (constraint stated in Section 5.2) and allowing us to implement quick migration experiments.

5.4.2 Architecture implementation

As general architecture, we propose to create a model that uses the COM model as a back-end as shown in the Figure 5.3.

We propose lazy access to the COM model back-end, which will guarantee that we access and load only what is needed. This feature aims to limit the memory usage (constraint stated in Section 5.2) by construction. The lazy approach will also allow us to map each binary model entity to a model-entity one at a time. We

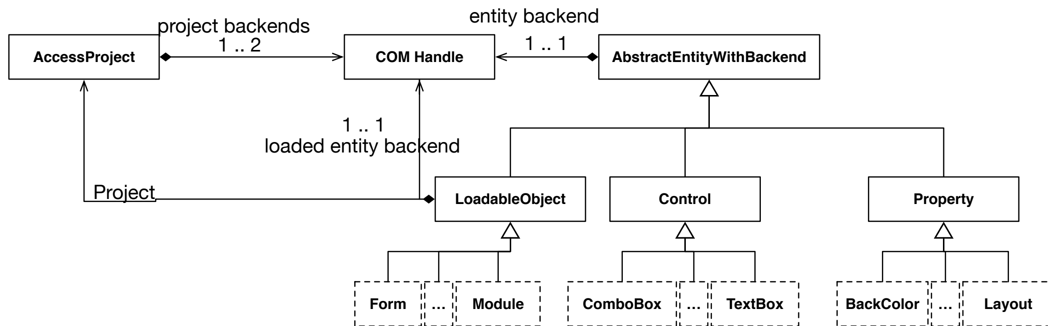


Figure 5.3: Architectural inbound design. The image shows the relationship between the different entities and their COM backend.

also propose to cache the results to reduce the COM calls and, therefore, CPU time and inter-process communication.

Regarding the mapping between the COM model entity type and our model, we propose to use two kinds of mapping: by type and by attribute value. Two COM models represent first-class citizen entities, and that is why all of them subclass from a *LoadableObject* class, which maps two COM models instead of one.

We propose to use factories to map the binary-model entities to model-entity types. The mapping factory by type maps one binary entity type to one model entity type. The mapping factory by attribute value maps one binary entity to one specific model entity type according to one specific binary-entity value.

5.4.3 Microsoft access model implementation

Our model extends from the architecture implementation and inherits the mapping to COM remote entities. This model is meant to be visited by a visitor pattern to perform analysis. It defines the structural components to be visited and relies on stateful traits. At the level of stateful traits, we define, by example, the widget-control composition.

5.4.4 Meeting the challenges and constraints

Now we revisit the challenges we faced (Section 5.3.3) and explain how we solved them.

The reading of a property of the COM entity, may give back another COM entity. Each model type must know which readings will give back a COM model entity. We use a factory that maps the COM model type with a model type in these reads. After creating a new instance, it sets up the given COM model entity as a back-end.

One model entity may correspond to more than one COM entity. There are two kinds of objects with more than one back-end, the first-class citizens, a subclass of LoadableObject, and AccessProject, which is a convenient class for managing the generality of COM usage. Since there are only two specific cases, they are treated individually.

One COM entity type may be mapped to different model entity types. While loading properties, we use a factory pattern that defines the class to instantiate according to the property's name. Since the only COM model entity with this kind of mapping is the property, we did not generalise this kind of mapping.

Loadable objects. For loadable objects, we defined a specific branch in the hierarchy, that before accessing remote properties related to the loaded object back-end, it ensures that the back end has been loaded and bound. To ensure this, the Loadable Object subclass does a typed, double dispatch with the AccessProject, which delegates to DoCmd.

Summary. Our approach addresses all the re-engineering challenges, responding *#RQ2: Are we able to re-engineer the meta-data sources into a valuable model for migration?*. It also limits computational resources usage, If we want to access all possible data, we risk having a model that is too big to be managed. To approach a solution to this problem, we propose to specialise the access on demand in our implementation by using *lazy loading* and *cache*. Lazy loading contains the memory usage to the effect of needed data. Cache scopes the inter-process communication and CPU time for remote access to one time per object.

5.5 Validation

In this section, we validate all the research questions by validating #RQ3: *Does the obtained model provide enough accurate information for conducting a language migration?*. Since our model is meant to be used for migration, we fully migrated some projects to the same technology. That is to say to replicate or clone. To do so, we perform a replication of ten different access projects. We use our model and the COM extensions for this performance to produce the replicated project programmatically.

5.5.1 Methodology

Chosen projects. For this validation, we used ten different projects (described in Table 5.1). Eight of them are base libraries used by Berger-Levrault in all the

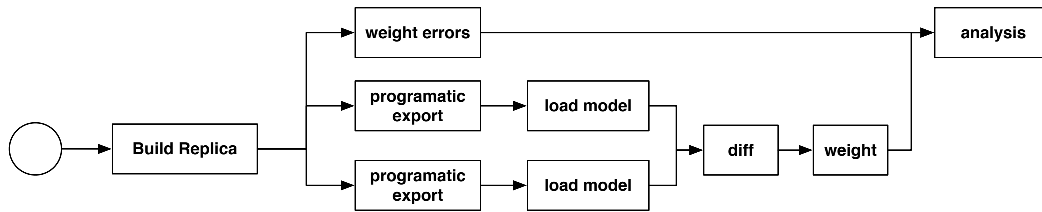


Figure 5.4: Validation methodology overview

Microsoft Access projects. One is an open-source example in GitHub [6]. The last one is the Microsoft Northwind Traders (Northwind for short) [7] example project. This project is used for learning Microsoft Access and uses most of the standard techniques and available graphical features of the language.

Table 5.1: Projects descriptions

Project	Remote Table	Table	Remote Query	Query	Module	Classes	Report	Forms
Northwind	0	20	0	27	6	2	15	34
CUTLCOMP	0	1	0	0	3	0	0	0
CUTL	7	3	0	1	26	62	0	8
CRIR	5	4	0	16	6	0	2	3
CPDI	0	1	0	0	2	0	0	0
CHABIL	11	2	0	27	8	1	1	10
CDDE	0	1	0	0	2	0	0	0
CAUNIT	0	1	0	0	4	15	0	1
ACCUEIL	25	7	0	13	6	67	5	33
Access Examples	0	10	2	14	11	1	8	13

5.5.1.1 Error tracking & weighting

Error tracking & weighting. All errors during the replication process are tracked for further analysis and correlation with the original/replica comparison. The error-tracking composition responds to the same composition as the proposed model. In a nutshell, we count the operations required to replicate the project. Each operation may succeed or fail. We count each of these results to compare.

Error tracking. COM is not intensively used to create projects programmatically. Many standard procedures, after running, make Access unstable and prone to fail in any next modification attempt. Failures that may imply from non-created widgets to missing properties. During the whole process, we track down all the errors that happened during this process to plot alongside the results of the comparisons. Each operation result is tracked down at the level of replication operation

[6]https://github.com/Access-projects/Access-examples
[7]https://docs.microsoft.com/en-us/powerapps/maker/canvas-apps/northwind-install

and typified: (1) ChildCreatedSuccessfully; (2) FailureToCreateChild; (3) FailureToWriteProperty; (4) PropertyWrittenSuccesfully. The tree of error tracking composition responds to the same composition as the proposed model.

Error weighting. We measure the failure of a replication process by the weighting and summarisation of the tree of operations.

Let o be the result of an operation of replication. Let c_o be the children's creation operation under the scope of the operation o. Let p_o be the properties creation operation under the scope of the operation o.

$$F(o) = \begin{cases} 1 & o \in \{Failure\} \\ 0 & o \in \{Success\} \wedge |c_o| = 0 \wedge |p_o| = 0 \\ (0.9\frac{\sum_{i=1}^{|c_o|} F(c_o i)}{|c_o|} + 0.1\frac{\sum_{i=1}^{|p_o|} F(p_o i)}{|p_o|})0.5 & o \in \{Success\} \wedge |c_o| > 0 \\ \frac{\sum_{i=1}^{|p_o|} F(p_o i)}{|p_o|} & o \in \{Success\} \wedge |c_o| = 0 \end{cases} \tag{5.1}$$

This recursive function calculates the proportion of errors in terms of composed errors. For our work, those elements composed of elements (for example, the controls inside a form) are mainly represented by their children. This is why one formula branch uses coefficients: 10% based on the component properties and 90% on the children's completeness. The 50% is because we consider that half of the thing is to create the element we are analysing.

5.5.1.2 Measureing completeness by file x file diff & weighting

Programmatic export for obtaining file definitions. We extract a model from file definitions to propose a verification that does not rely on our COM model. To obtain each component's file definition, we leverage the COM function named SaveAsText (explained in Section 5.7). The output of this function differs by each type of entity as follows: For Modules and class modules, it offers VBA (.bas) files; Queries offer an SQL output; For tables, it uses XML format. Forms and Reports offers an output that hybridises the structure definition and the VBA code of the companion module.

File diff & weighting. We instrument a diff between pairs of files of each original / replica exported project. The result of this diff is a differential graph expressing all the required operations for transforming the original project into a replica project. We count the comparisons and contrast according to the outcome: added/removed element, modified value, an exact replication. By counting the differences, we

measure the completeness of a replica: if the replica is identical to its source, it is complete.

File diff. To be able to diff each pair of files, we used different techniques Modules, Macros, and Queries are loaded as nodes with names and plain text content. Tables are loaded as XML trees, including names, indexes and fields with their name and type. Forms, Reports are loaded with a simple parser that produces a tree of reports/forms with their controls and properties.

Each of these entities is loaded from the original and replica. For each pair, we calculate the differential tree expressing all the needed operations for transforming the original graph into the replica graph. We define the following operations: (1) Add (2) Remove (3) Same (4) ModifyChild (5) ModifyProperty.

Diff weighting. We measure the completeness of each of the elements on the differential graph. Let u be the result of comparing an element from the original project with its equivalent of the replica.

$$Completeness(u) = (1 - M(u)) * 100) \tag{5.2}$$

Magnitude $M(u)$ is the weighting of the difference between two elements. Let u_o and u_r be, respectively, original and replica sides of u. Let c_u be the set of children that belong to the u. Let p_u be the set of properties that belong to the u.

$$M(u) = \begin{cases} 1 & u \in \{Add, Remove\} \\ 0 & u \in \{Same\} \\ (0.9\frac{\sum_{i=1}^{|c_u|} M(c_u i)}{|c_u|} + 0.1\frac{\sum_{i=1}^{|p_|} M(p_u i)}{|p_u|})0.5 & u \in \{ChildModif\} \wedge |c_u| > 0 \\ \frac{\sum_{i=1}^{|p_u|} M(p_u i)}{|p_u|} & u \in \{ChildModif\} \wedge |c_u| = 0 \\ u_r - u_o & u \in \{PropertyModif\} \wedge u_r, u_o \in \{Native\ type\} \end{cases} \tag{5.3}$$

This recursive function calculates the magnitude of the difference in terms of the composed differences. The coefficients in this formula respond to the same explanation as those used in the error weighting formula.

5.5.2 Results

Table 5.2 offers an overview of the replication process of each of our projects. Most of the main elements are replicated. The 48 tables and 2 queries not replicated are remote entities that we cannot access since we lack access to the remote server. This account for those tables and queries

Table 5.2: Export overview

	Reference	**Table**	**Query**	**Module**	**Report**	**Forms**	**Total**
#Elements	70	98	100	222	31	102	623
#Replicated	69	50	98	222	31	102	572
#Failures	1	48	2	0	0	0	51

Below compare the different aggregation of completeness with the aggregation of error tracking per project per element type. We do not include modules nor macros because the result is **complete** in all the projects. Since both completeness and error series respond primarily to a hyperbolic distribution, we propose measuring the rate of success and error using the median.

Table 5.3 show some very good results. We fully replicate most of the queries. Table 5.4 also has excellent results since most of the failures happen on remote tables, those projects are: **CUTL**, **CRIR**, **CHABIL**, **ACCUEIL** and **Access Examples**. These projects have remote tables (these tables cannot be cloned at all, implying full losses: 100), which contrasts strongly with the minimum failure rate (0), yielding large standard deviations(50.63). There are some cases where there are no errors during the process, but we don't meet full completeness, such as the cases **CUTLCOMP**, **CAUNIT** and **CPDI**. These cases happen because system tables are not replicated, and the replication targets a newer Microsoft Access version, holding different system tables.

In the particular case of reports and forms (Table 5.5, Table 5.6), we see less interesting outcomes from the point of view of completeness, but we observe an inversely proportional relationship with the errors. There is also a restriction of implementation, many values are stored in byte array structures, and even if we can read them, we cannot write them. This makes replicating printing configuration, custom controls based on ActiveX or OCX technologies and image contents impossible. These properties do not figure in the errors because the construction of the process avoids them. Finally, Figure 5.5 and Figure 5.6 provide an idea of the confidence interval of the measures. Both sides show a correlated existence of scattered measures. The completeness confidence interval is too large in the case of the *Tables*. We can relate it to the scattered error measures. Forms and Reports completeness show shorter intervals that can correlate with the error intervals and the distance between the isolated cases. Finally, modules and queries have a really good interval. The centre is almost 0 in the failures plot, even with some isolated cases.

5.5.3 Human insight and opinion

We check each of the replicas and compare them manually with the original and also with the extracted data. Most of the significant parts of the applications were

Table 5.3: Query comparison

Queries								
Projects	**Completeness**				**Failures**			
	Max	Min	Median	SDev	Max	Min	Median	SDev
Northwind	100	100	100	0	15	0	0	3.05
CUTLCOMP	–	–	–	–	–	–	–	–
CUTL	100	100	100	0	0	0	0	0
CRIR	100	88.46	100	2.88	0	0	0	0
CPDI	–	–	–	–	–	–	–	–
CHABIL	100	93.49	100	1.25	0	0	0	0
CDDE	–	–	–	–	–	–	–	–
CAUNIT	–	–	–	–	–	–	–	–
ACCUEIL	100	95.76	100	1.17	0	0	0	0
Access Examples	100	0	100	25.81	100	0	0	34.15

Table 5.4: Table comparison

Tables								
Projects	**Completeness**				**Failures**			
	Max	Min	Median	SDev	Max	Min	Median	SDev
Northwind	100	0	100	23.72	22	0	0	5.14
CUTLCOMP	100	0	100	27.62	0	0	0	0
CUTL	100	0	98	5041	10	0	0	46.43
CRIR	100	0	99	46.04	100	0	0	44.42
CPDI	100	0	100	27.62	0	0	0	0
CHABIL	100	0	0	50.65	100	0	0	50.38
CDDE	–	–	–	–	–	–	–	–
CAUNIT	100	98	100	0.75	0	0	0	0
ACCUEIL	100	0	0	49.6	100	0	100	50.63
Access Examples	100	0	99.5	32.34	90	0	0	17.65

Table 5.5: Form comparison

Forms								
Projects	**Completeness**				**Failures**			
	Max	Min	Median	SDev	Max	Min	Median	SDev
Northwind	86.24	61.55	69	4.72	8.68	5.32	9	0.73
CUTLCOMP	–	–	–	–	–	–	–	–
CUTL	86.73	63.46	82	8.45	7.62	1.66	5	1.79
CRIR	74.73	73.06	74	0.96	8.31	8.18	9	0.064
CPDI	–	–	–	–	–	–	–	–
CHABIL	78.3	65.89	70.5	4.14	8.95	5.44	6	1.134
CDDE	100	98	100	0.83	0	0	0	0
CAUNIT	79.96	79.96	79.96	0	5.27	5.27	5.27	5.27
ACCUEIL	92.71	70.01	78	6.22	15.71	2.49	6	2.7
Access Examples	88.52	61.91	76	8.35	34.05	6.6	9	7.25

Table 5.6: Report comparison

Reports								
Projects	**Completeness**				**Failures**			
	Max	Min	Median	SDev	Max	Min	Median	SDev
Northwind	71.55	65.97	70	1.5	16.57	11.33	16	1.3
CUTLCOMP	–	–	–	–	–	–	–	–
CUTL	–	–	–	–	–	–	–	–
CRIR	73.13	71	73	1.5	20.64	15.64	18.5	3.54
CPDI	–	–	–	–	–	–	–	–
CHABIL	71.21	71.21	71.21	0	13.87	13.87	13.87	0
CDDE	–	–	–	–	–	–	–	–
CAUNIT	–	–	–	–	–	–	–	–
ACCUEIL	74.57	73.34	74	0.53	14.16	13.87	14	0.13
Access Examples	90.14	66.1	72.5	7.48	18.15	13.87	16	1.51

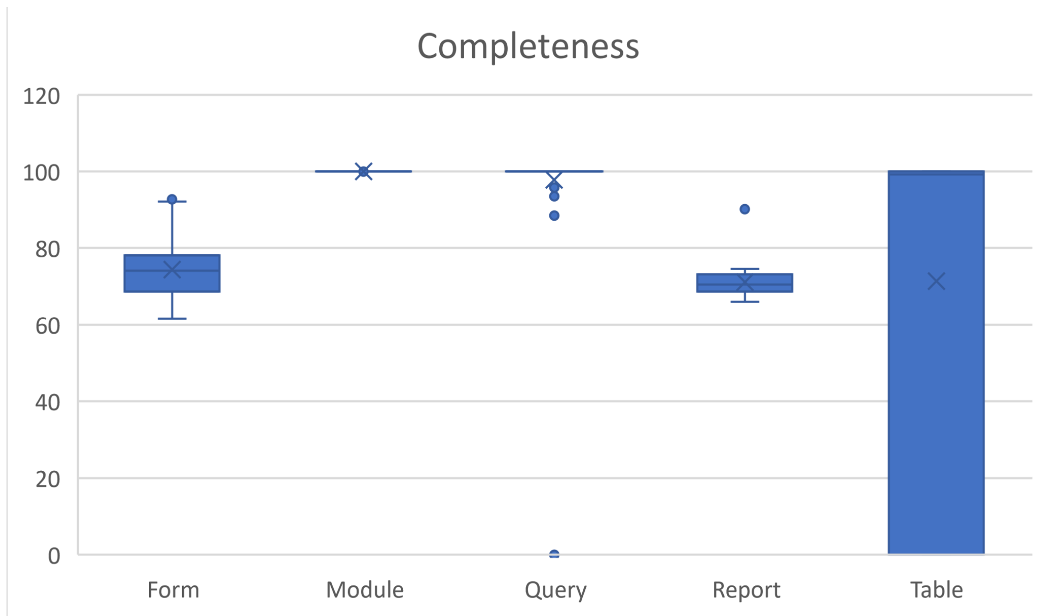


Figure 5.5: Completeness Confidence Intervals.

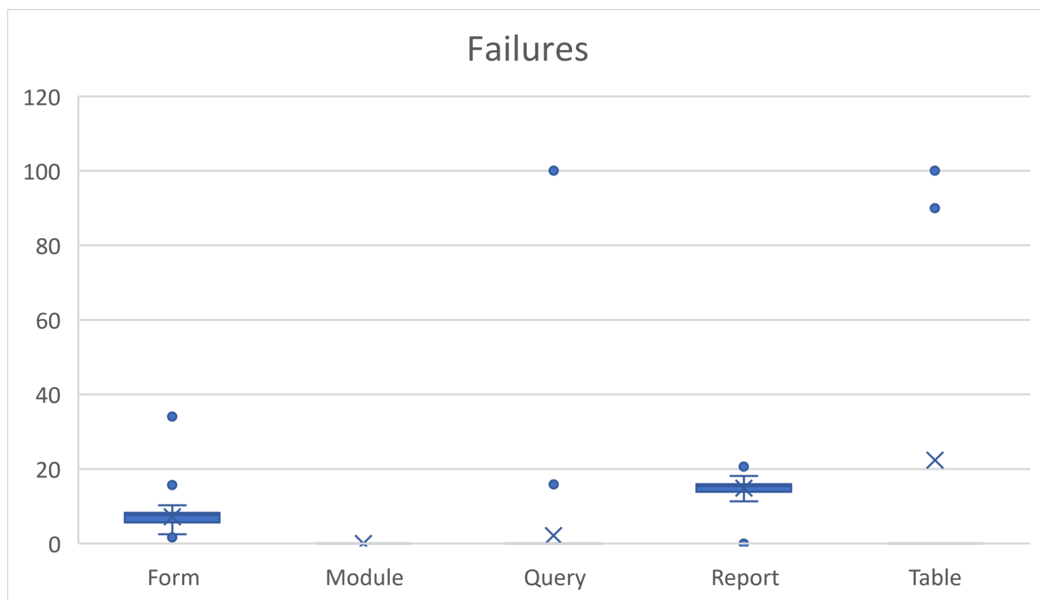


Figure 5.6: Failures Confidence Intervals.

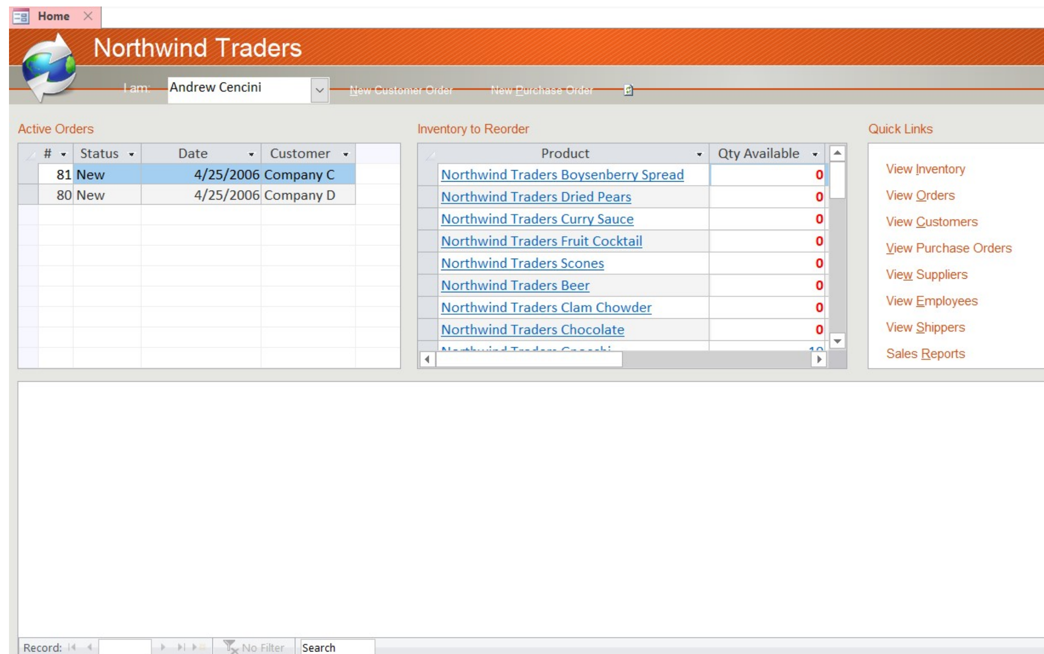


Figure 5.7: Original Home screen

replicated correctly, even when some of the most appealing graphical features were not maintained because of the impossibility of writing this kind of data using COM. Nevertheless, we spent time, especially on the execution of many functionalities of the project Northwind and found out that most of the behaviours are maintained since all the macros and source code has been correctly replicated and bound to the proper structures. After migrating all the example data available from the original to the replica, we can observe that the login works as expected in all the tests we manually checked. Figure 5.7 and Figure 5.8 give material evidence of the outcome by exposing the most complex form in the replicated system. All this insight is highly positive. Our most positive but opinionated insight is that we developed the validation faster than expected, thanks to the model we are presenting. We got a constant assessment from it getting fast feedback and understanding of the replication process target.

5.6 Threats to validity

Internal threats to validity. *Face validity.* Our validation is based on the replication of ten projects. This gives about 534 first-class-citizen components, thousands of controls and table fields. Many kinds of components may not be represented by those we have.

Construct validity. We have seen how the non-replication of system tables available in other versions of Microsoft Access came out as a difference between

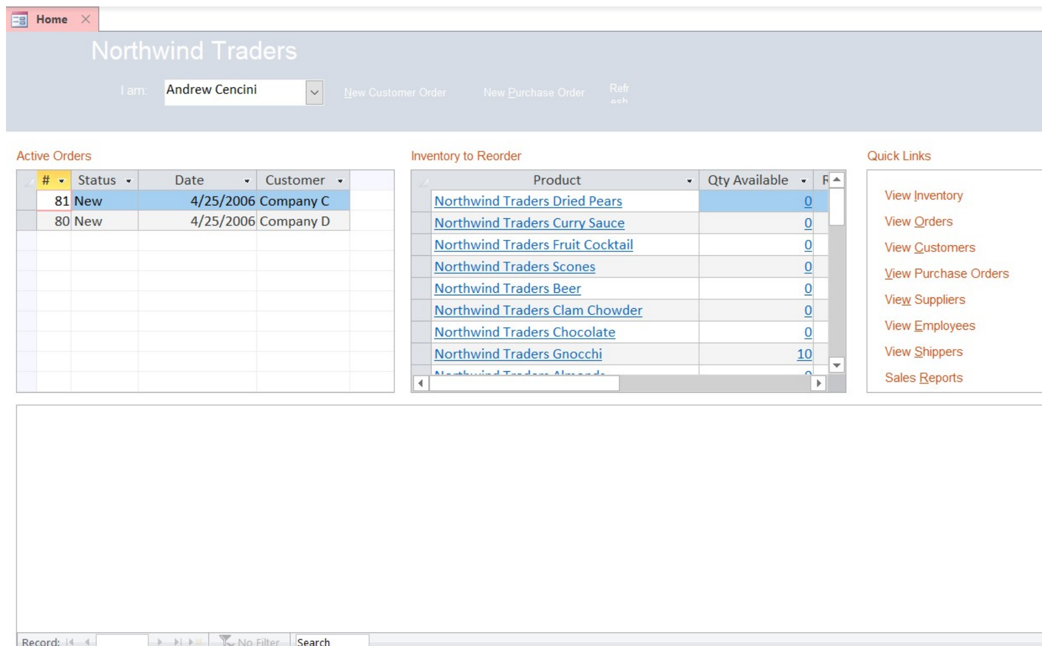


Figure 5.8: Replicated Home screen

the original and replicated code because of the policy of non-exporting system tables. This does not happen to be a false positive. And we did not find any false positives or negatives, but we cannot ultimately ensure that this cannot happen in other projects.

External threats to validity. *Temporal validity.* We had to leverage some undocumented functions to allow the file comparison for our validation (widely explained in Section 5.7). This function could change or not be accessible in newer versions of Microsoft Access, threatening the reliability of the process.

5.7 Discussion

Programmatic exporting solution. As we pointed out in Section 5.2, there are third-party solutions for source control that could help solve the opacity problem. Control version software produces different text formats able to reproduce the same project. The available tools developed for source control are based on the usage of **SaveAsText** undocumented function provided by the Microsoft Access DoCmd command. This function exports each entity to a text format, producing XML for the tables, VBA for the source code and a Microsoft Access-specific DSL for defining forms and reports. Their stability is already tested for many years by the market, which could be a good starting point for software analysis.

Undocumented features. Even considering that these tools have been in the market for a long time, we do not know how they will change in the future. As we pointed out, these solutions use the undocumented functions SaveAsText and LoadFromText. This presents two risks: (1) Microsoft may change their behaviour or even make them unavailable in the future, and (2) the format of the exported text has no documentation either, which means that different versions could have singularities. The directly threatens any tool we want to produce on top of our model.

Context and performance. Our approach to software migration is based on tooling the developer. For this reason, it is more beneficial to see exactly what the developer is actively working on to have more context and insight. This is not possible by analysing text but by analysing a working environment. Finally, we see that delegating the management of the information to the same access and using the IDE as a database has a high potential for reducing memory consumption and model complexity, allowing us to develop tools that can run in a working environment without requiring extra infrastructure. We only load the information that is used.

What our validation does not validate. The exported files have minimal information required to build the whole application again. However, it does not include default values. While in a file, a complex component may define about ten different properties, when accessed through COM, we have access to more than 100 properties per control. This means that the text representation is incomplete, *partially observable*. From the point of view of software analysis and migration, having systematic access to default values without having to specify them manually is a great asset.

5.8 Conclusion

In this chapter, we explained the problem of opacity in Microsoft Access. We propose an approach that enables modelling based on the Microsoft Access COM API. We validate that it is possible to create a model out of this API and propose a model that loses little relevant information in specific cases. However, this model is only able to represent Microsoft Access applications. In the next chapter, we will study how to model multiple applications of different technologies with the same meta-model.

CHAPTER 6

Heterogenous Unified Meta-Model: Model migrating systems

Contents

6.1 Modelling **72**

6.2 Understanding language migrations **72**

6.3 Heterogeneous Unified Meta-Models **75**

6.4 Discussion **79**

6.5 Conclusion **80**

This chapter introduces a model for migrating applications based on conceptual likeness and unlikeness.

Section 6.1 discusses the need for a modelling approach focused on migrating applications. Section 6.2 introduces a simple example of language migration followed by the challenges that must be faced by a model responding to our problem. Section 6.3 proposes the approach of an *Heterogeneous* Unified Meta-Models (HUMM).

We note that the content of this chapter is based on our article on submission "Understanding the Migration of Applications with Typing Ontologies" Bragagnolo *et al.* [Bragagnolo b].

6.1 Modelling

In Section 4.2.2, we explicitly remarked on the need for models. We also discussed their shortcomings. We find out two families of meta-models. Those representing a single programming language, such as Famix, and those representing a paradigm, such as UML.

Single programming language meta-models do not share entities with other meta-models. A side effect of this rigidity is to make language migrations harder. While this rigidity helps maintain the correctness of a model and provides essential functionalities such as querying, it prevents the reuse of the meta-model and the comparison of model entities, complexifying the act of migration. For example, an IF statement of a Java language meta-model cannot be used in a Typescript language meta-model.

Paradigm meta-models represent only the paradigm concepts. We name these meta-models *Homogeneous Unified Meta-Model*, as they homogenise and smooth the edges of implementation to ease the understanding of the design and structure of a program. However, a paradigm is an entelechy that responds to no computational restriction, while programming languages implementation must take technical decisions that shape these concepts differently. This implies losing unique semantical values, which are essential as discussed in Chapter 3. For example, Terekov *et al.* [Terekhov 2000] presents the case of two different versions of COBOL, where the same type represents a different number of decimal digits, leading to problems representing money.

This chapter presents a Heterogeneous Unified Meta-Model, representing common concepts only once, allowing developers to define unique extensions for each language.

6.2 Understanding language migrations

To understand what we have to model, we must first understand what language migration is, which is one of the many migrations we must conduct, requiring more detail.

6.2.1 Language Migration

Migrating an application from a source language to a target language implies we must produce a new application with the same semantics as the original application but in a different language/technology. Such migrations are challenging because different languages propose different concepts, some unique to the language *e.g.* Java generics, MS Access error management or Pharo `thisContext`. However, programming languages also present overlaps in common concepts such as

functions, methods, and classes. To migrate means at least understanding the differences between the source and target language.

6.2.2 A language migration example

Let us consider the module Listing 6.1. This module contains a factorial function written in Visual Basic for Applications (VBA).

```
Function factorial (i as Integer) as Integer
   If i = 0 Then
     return 1
   Else
     return i * (factorial( i - 1))
   End If
End Function
```

Listing 6.1: VBA Factorial Function

Migrating this module to Java through the application of successive transformations raises questions for the developer: What are the differences between source and target languages? Which transformations to implement? If we disregard the details in style, such as how to declare a variable (`i as Integer` in MS Access, `int i` in Java), migrating this example function requires the following three transformations:

1. transforms the MS Access module into a Class,
2. transforms the function into a static method,
3. change the type from Integer to int and
4. transforms the `factorial` invocation into a static method invocation of the form Class.factorial().

The proposed transformations address mainly the change of paradigm and type. The outcome of this transformation yields

```
Class MyClass {
  static int factorial (int i) {
    if (i == 0) {
     return 1
    } else {
     return i * (MyClass.factorial( i – 1))
  }
}
```

Listing 6.2: Java Factorial Method

6.2.3 Problem and challenges

Modelling migrating programs is expensive and challenging.

6.2.3.1 The cost of representing different languages

Traditional approaches for software migration use different meta-models to represent the source, and target languages [Kontogiannis 1998,Zou 2001], or use a single model for the source language and produce a string for the targets [Brant 2010]. On the one hand, such meta-models duplicate the overlapping concepts between those languages. On the other hand, systematic unification would prevent the correct representation of features unique to a language, either by computational unicity (*e.g.* the order of execution of arguments) or by unique existence (*e.g.* any of the infrastructure-based statements and libraries provided by fourth-generation languages such as MS Access).

6.2.3.2 Representing *inconsistent* intermediate states

The example in Section 6.2.2 presents three transformations that, when successively applied, produce a Java equivalent piece of code. However, the described program is either wrong or incomplete in between transformations. For example, once the module is transformed into a class, it incorrectly contains a function. It is worth noticing that this problem will remain regardless of the chosen order. Thus, to correctly support migrations, models must admit inconsistent intermediate states without losing information.

6.2.3.3 Assessing model correctness

We want both: to allow inconsistent intermediate states and to tell if a model is or is not correct. To do so, we must be able to tell if a model is correct according to a given language. For example, the meta-model has Function and Module classes, which we use to represent Access functions and modules. The meta-model also has Method, AnnotatedMethod, ParametrizableMethod and JavaClass classes, which we use to represent Java classes and different methods. As we just pointed out before, we need instances of JavaClass to be able to hold instances of Function as a method to allow inconsistency. Still, we need to be able to tell that this relation is wrong because instances of JavaClasses can **only** hold as methods instances of the Method, AnnotatedMethod or ParametrizedMethod.

6.2.3.4 Planify a migration

Many articles have been written about the different mappings between languages [El-Ramly 2006, Trudel 2012, Martin 2002b, Malabarba 1999, Kontogiannis 1998, Mossienko 2003]. However, the process of extracting and understanding those

mappings requires manual analysis. There is a need to understand better the non-overlapping concepts between two languages and their instances inside a particular code model. Non-overlapping concepts represent semantic differences between languages and highlight the significant effort of migration. Moreover, understanding the instances of non-overlapping concepts in a particular application would help plan the order to perform the migration.

6.3 Heterogeneous Unified Meta-Models

In this chapter, we study the usage of heterogeneous meta-models to represent migrating applications. A heterogeneous meta-model represents common concepts only once, allowing developers to define unique extensions for each language.

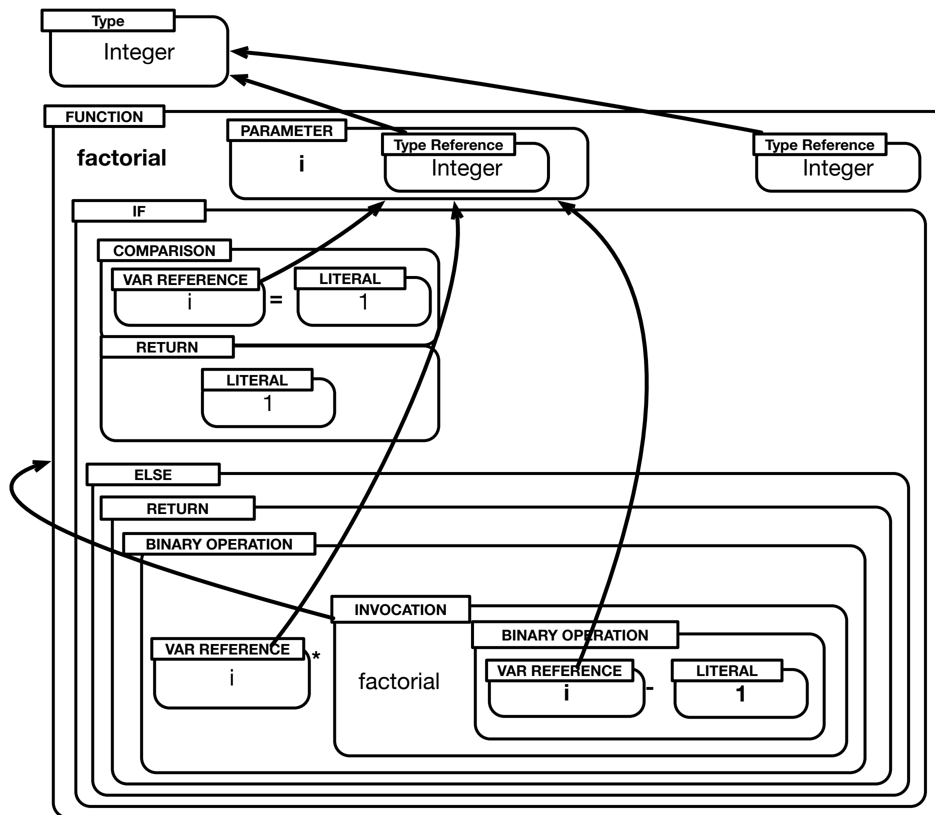
6.3.1 Heterogeneous Abstract Semantic Graph

We propose using an Abstract Semantic Graph (ASG) – presented before in Section 4.2.2. Figure 6.1a shows the ASG that represents the piece of code listed before in Listing 6.1. Figure 6.1b shows the ASG that represents the piece of code listed before in Listing 6.2. Both ASGs are highly similar. Each node in the graph is represented as a box; its composition is represented by including a box inside another box. Each pattern used for drawing the box indicates a different kind of object in terms of role. The only arrows drawn are the ones that turn it into an ASG.

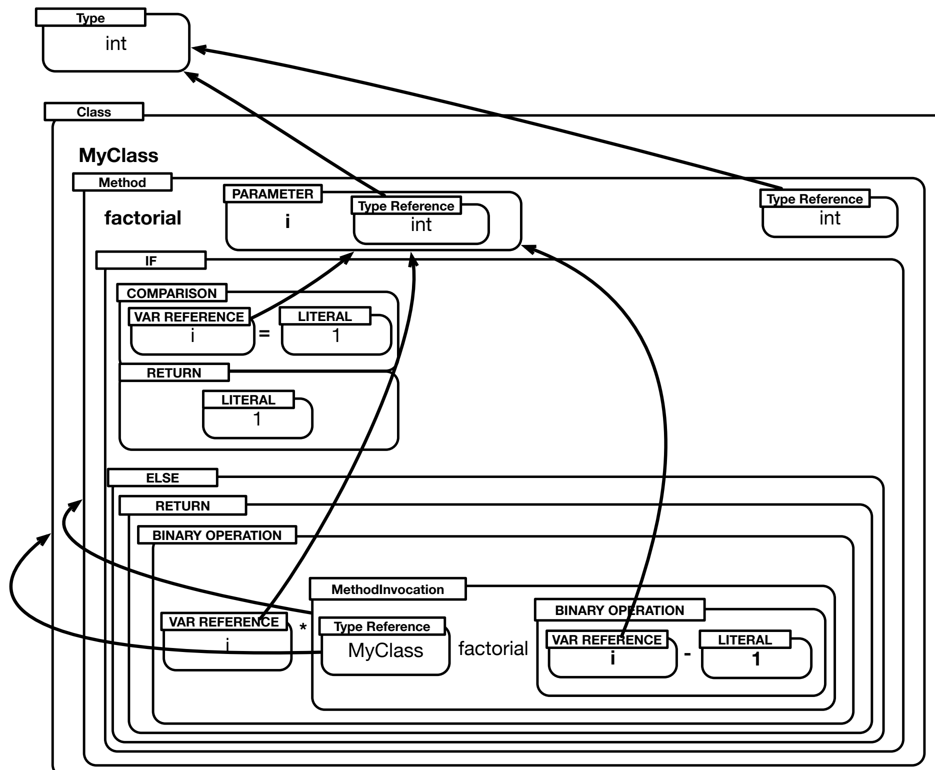
We use a single HUMM to reify each concept of each different language. Using the same meta-model enables the unification of all the equivalent concepts from the point of view of migration and the creation of new concepts when it's not the case. For example, a single entity in our meta-model represents an IF statement or the reading access to a variable. Our meta-model also defines specific entities for concepts that are somehow similar but have semantic differences in each language *e.g.* For example, classes in Java can be defined within another given class, while in Typescript, this is not allowed.

The meta-model offers entities that make sense or do not according to the language. This allows us to represent many languages with the same meta-model. However, not all entities are used in all languages. We call our approach Heterogenous to emphasise its diversity.

The basic structuring concepts From a grammatical and syntactical point of view, we often find three main concepts: Declarations, Statements and Expressions. A *Declaration* is any construction that structures and names an artefact. *e.g.* Type, Class, Module, Function, etc. A *Statement* is any construction that structures and carry on some behaviour. *e.g.* control flow structures such as if, while, etc. An *Expression* is any construction that carries out some behaviour and yields a value. *e.g.* Assignment, built-in operators, primitive call, function invocations, etc.



(a) Microsoft Access ASG Model example. Box composition shows element inclusion. Arrows highlight use.



(b) Java ASG Model example. Box composition shows element inclusion. Arrows highlight use.

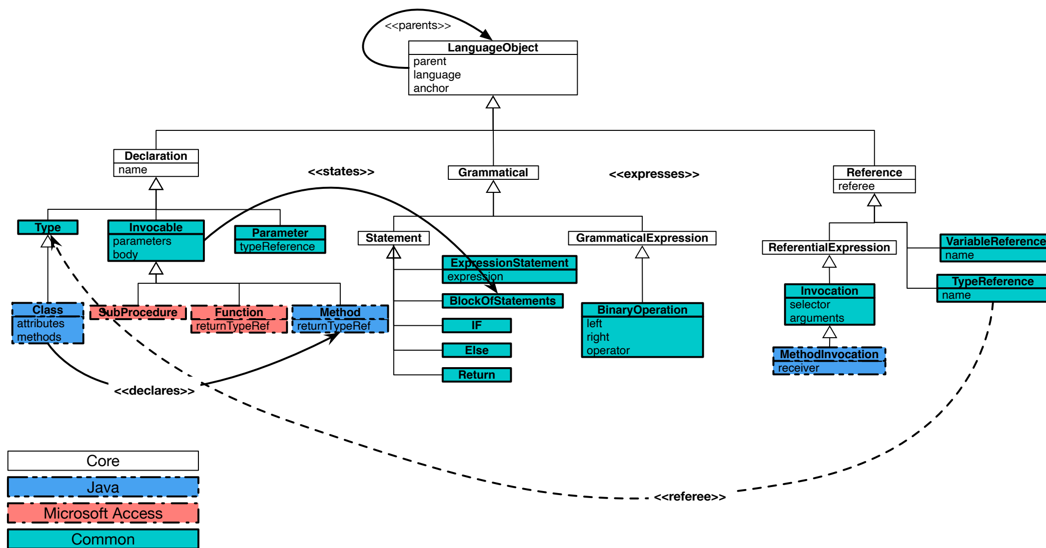


Figure 6.2: Meta-model hierarchy used to represent the Java and Microsoft Access ASG examples. Our entities mainly inherit from Declaration, Reference and Grammatical. They relate using typified relations.

Since we add the linking dimension to jump into ASG structure, we take into account a fourth concept: the Reference.

A *Reference* is any construction that refers to some artefact defined elsewhere. *e.g.* Function invocation, message-send, typed variable declarations, etc. In contrast with the reference, we add *Grammatical*, any construction that bases its behaviour on its grammatical structure instead of a referred artefact.

Figure 6.2 shows a simplified and incomplete meta-model able to represent Java and Access. In this meta-model,s we find each of these core elements. All the core elements inherit from LanguageObject. LanguageObject class defines default behaviours and minimal structures: All entities in our model have a parent and a language. The parent indicates where the node was declared. It reinforces the hierarchical nature of our structure. The language is used to declare the element's language and is used by the element as an oracle to know if a relation between this and another object is incorrect.

In Figure 6.2, we also find the meta-model entities required to represent the factorial ASG in Java and Microsoft Access. We observe that a large majority of meta-model entities are used for both languages, but some of the entities are only used in one language.

6.3.2 Relations

In our model, we typify the relations to understand the nature of the relation between concepts. We recognise seven kinds of relations: Parent, Declares, States,

Express, Refers, Referee and Property. None of these relations entangles any static typing.

Parent This relation is expected to relate an object with the object where the first exists: defined, declared, used, etc. For example, a method can have a class as a parent. A binary expression is the parent of the left-hand and right-hand expressions.

Declares Describes the relation between a "declarer" entity (often a declaration that works as scope) and a "declared" entity. A class declares methods. It is the opposite relation to the parent relation.

States Describes the relation between a "stating" entity (often a declaration that works as scope) and a "stated" entity. A function states a while statement. It is the opposite relation to the parent relation.

Expresses Describes the relationship between an "expressing" entity and an "expressed" entity. *e.g.* an "if statement" expresses a condition. It is the opposite relation to the parent relation.

Refers Describes the relation between a "referrer" entity and a "referring" entity. A variable declaration refers to a *type reference*.

Referee Describes the relation between a *"referring"* entity and a *"referred"* entity. A *type reference "referee"* a type. A *function invokation "referee"* a function.

Property Describes the relation between an entity and any other terminal object. *e.g.* A class has a property "name" with the name of the class. A function has a selector.

6.3.2.1 Relations correctness

A relation must be able to ensure correctness at runtime. The correctness of a relationship relies on the relation kind, the interrelated entities, and the language. This is why all our entities have a language object at hand, for the relations to delegate the application of constraints on a language object. This allows our model's relations to accept or reject objects according to the language. A Class object may accept to have "functions" in one language and strictly refuse it in another language.

6.3.3 Language reification

Our model must support many languages. We propose reifying the language as an object to deal with each language's specificities. This object has multiple responsibilities. One of them is to tell if a HUMM is correct according to the language. To tell if a model is or is not correct, a language object can be configured with one of three different strategies:

(i) Permissive: all relations are allowed. (ii) Structural-Role: allows any relation established by role. For example, the *"declares"* relation requires the declared

object to be a declaration. (iii) Ontological: allows any relation a typing ontology allows, used as an oracle [Bragagnolo b].

6.4 Discussion

We avoid modelling each programming language independently, trying to have, as much as possible, all language ASGs represented using the same types. In this, we diverge from what a meta-model like Famix does (one specific meta-model for each programming language [Anquetil 2020]) and are closer to what KDM of the OMG proposes (one unique meta-model [OMG 2011]). Our rationale for having only one meta-model is that when migrating from one language to another, we want to keep as much of the source ASG as possible. Having the same nodes for both languages helps us do that as it allows us to "migrate" by just copying an entire branch of the ASG (e.g. a complex numerical expression) from the source model to the target one. This allowed us to use the same rules for this experiment when migrating to Pharo and Java.

Yet, it is not always possible to do it because, as noted in [Anquetil 2020] "various programming languages have minor semantic differences regarding how they implement [some programming concepts]". The same goes for ASG nodes; for example, some languages have an "elseif"[1] statement (or part of the statement), and others represent it as an "else" containing a new "if". (Note that KDM or GASTM do not offer an "elseif" node.)

To accurately migrate a language, these minor semantic differences must be modelled. Therefore, we use the same ASG nodes for different languages as long as the nodes do have the same semantics. When this is not the case (*e.g.* Java classes and TypeScript classes), we create different ASG nodes to be able to differentiate them in the migration rules.

6.4.1 Model Integrity

Due to our modelling choices, our meta-model favours permissiveness over semantic integrity. For example, we will allow all our "if" statements to have an "elseif" part even if this is not true for a language.

However, to be able to tell at any moment what entities in a given model are or are not correct, we reify the typing restrictions of a model concerning a given language by automatically extracting a domain ontology representing the typing constraints of the given language. Like this, using instances of JavaClass is only allowed in a model where the entities are configured with the Java language reification. [Bragagnolo b].

[1] For example, "elseif" in Visual Basic or PHP, "elif" in Python

6.5 Conclusion

This chapter proposes a novel modelling approach specialising in migrating systems. We described and implemented a HUMM. This model is mainly used in Part III.

The next part of the thesis deeps into the understanding of a software migration, ponderating complexity and risks in software migration and helping to visualise the architectural complexities that must be faced to achieve a successful migration.

Part II

Understanding and Measuring Software Migration

CHAPTER 7

Mind the gap: Understanding Migrating Software

Contents

7.1 The need to understand 84
7.2 The Alce migration analysis tool 84
7.3 Architectural analysis 85
7.4 Risks and metrics 88
7.5 Visualisations 92
7.6 Alce exemplified on the *ePaie* project 98
7.7 Discussion 101
7.8 Conclusion 102

In this chapter, we present our tool for understanding software migration. For doing so, we also present metrics and visualisations aiming to represent most of the critical factors of migration like ours, based on our own migrating experience and those presented by the literature.

The chapter discusses the need to understand the migrating software in Section 7.1. Section 7.2 presents Alce: an analysis tool for predicting migration complexity. Section 7.4 presents the supported metrics, their visualisations and what they reveal of the migration. Section 7.5.2 presents the architectural visualisations and how the architecture constraints the migration. Section 7.6 presents the process used during the reporting for migration planning of an industrial project. Section 7.7 discusses the limitations of the approach.

We note that the content of this chapter is based on our article "Risk and Complexity Assessment on the Context of Language Migration." Bragagnolo *et al.* [Bragagnolo 2021b], and in the technical report "Alce: Predicting Software Migration." Bragagnolo *et al.* [Bragagnolo 2023],

7.1 The need to understand

Shortly after starting the research project, we met with the key developers and managers of the teams working on systems requiring migration. In this first meeting, the first question that popped up was: “How much time is it going to take to migrate this application?”. We cannot answer this question. However, we can provide enough data to allow team leaders and managers to estimate it based on arguments and an understanding of the dimensions of the projects.

We note that the literature does not systematise or measure migration projects but actively pinpoints specific challenges during different kinds of software modernisation, particularly migration (see Chapter 3).

While implementing our various tools and according to the literature, we proposed four metrics to understand the complexity of a language migration. We also propose two visualisations able to prioritise and express aggregated and individual features of the architecture of the source application. Along with these, we propose an analysis process used to gain insight into the software migration’s complexity, enabling the proposal of documented project blueprints and planning a software migration.

To provide enough data to allow team leaders and managers to estimate it based on arguments and an understanding of the dimensions of the projects, we decided to implement the visualisations we contributed to in this article. We have used these visualisations for writing estimations, planning reports and constructing the blueprints of an industrial project that requires multiple migrations, including architectural.

For implementing our approach, we leverage the model presented in Chapter 5 to be able to produce a Famix model [Anquetil 2020] for Microsoft Access [1], which allows us to use and extend the Moose platform [2].

7.2 The Alce migration analysis tool

A migration like ours involves a large project and an extensive process. Therefore we argue that stakeholders require assistance understanding the source system and the migration process. This process is tightly related to the source system, the target technology and the stakeholder’s expectations. We claim that analysing a system to understand the cost and complexities of migration requires more than just analysing the project as we are used for software maintenance [Bragagnolo 2021b]. We implemented Alce Analysis Tool [3] for that. Figure 7.1 depicts the general view of our tool.

[1] https://github.com/impetuosa/alce
[2] https://modularmoose.org/
[3] https://github.com/impetuosa/alcides

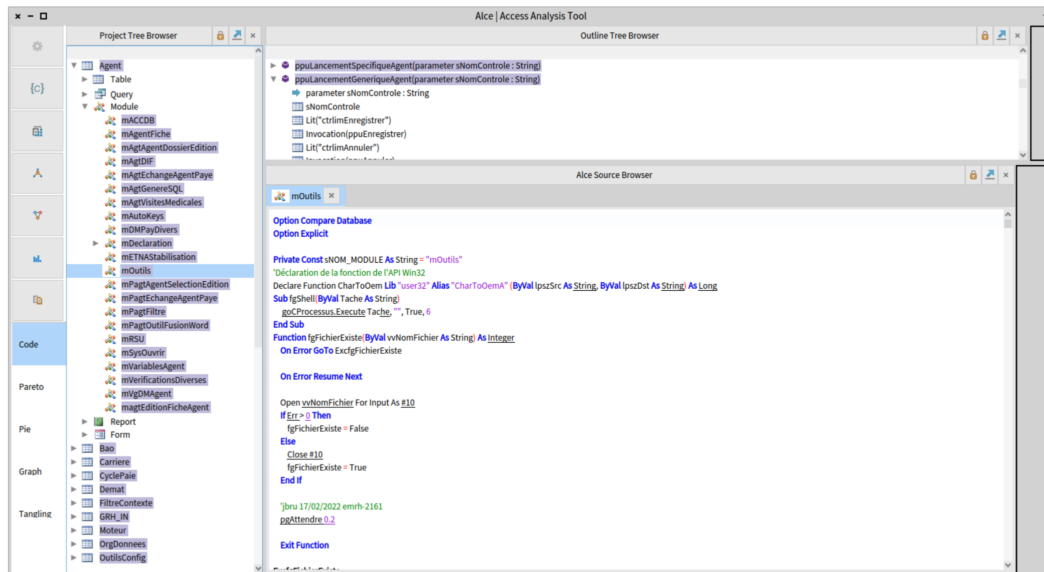


Figure 7.1: Tool Screenshot: From the left to the right, top to bottom, Toolbar, Project Tree Browser (to navigate the project), Outline (To navigate the elements defined within the selected element), Source Browser (to read and navigate the source code of the selected element).

7.2.1 Tool presentation

We implemented a tool to gather multiple aspects of the migration complexity to help Berger-Levrault's decision-making process during migration. This tool leverages existing Moose [4] infrastructure and know-how to create visualisations that help us understand the complexity of migrating projects.

On the left of Figure 7.1, we see a toolbar that lets the user choose a specific visualisation. Once the visualisation is open, we can observe it on the spot or slide it into our tool's right part. On the left of Figure 7.1, we find the Project Tree Browser after the toolbar. This widget allows the user to navigate and contextualise the opened visualisations. Selecting an item in this widget triggers the update of all the visualisations, except those that are locked, to avoid updating.

7.3 Architectural analysis

Forms, Reports, Modules and Classes in Microsoft Access can implement multiple concerns such as GUI, GUI navigation, data accessing, interoperability with other systems, and business logic. Knowing that the migration project aims to

[4] Moose is a platform for software analysis. It allows the representation software system in a model to query, manipulate, transform, and visualise this model. Moose is based on Pharo and is open-source software under BSD/MIT.

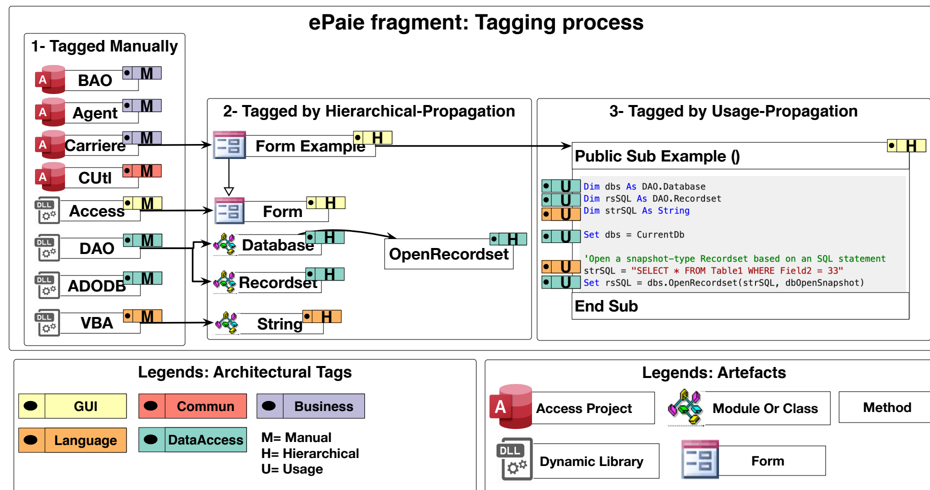


Figure 7.2: The tagging process: a fragment of the ePaie project being tagged. The figure is split into three parts: 1- Manually tagged, 2- Hierarchical-Propagation Tagged and 3- Usage-Propagation Tagged.

split the applications into at least front-end and back-end applications, we present a method to recognise architectural concerns and a graph that reveals the concerns of a selected artefact. To recognise the concern which belongs to each library or sub-project, we label the different artefacts with the name of the architectural concern—for example, GUI, DataAccess, Interoperability, Business, etc. We call them architectural concerns because they define to what architectural layer an entity belongs. The action of labelling an artefact is also known as tagging.

7.3.1 Tagging artifacts

One of the most complex obstacles to software migration is the tangling of architectural concerns: the spaghetti code. We expect the different artefacts to be tagged to enable automatic architectural analysis. Each artefact must be labelled with its architectural concern.

We ask the user to label the different libraries and sub-projects with their related architectural concerns.

To ease the task of tagging, we contribute two tag propagation algorithms: one for the declaration elements and the latter for the reference elements (see Chapter 6 for declarations and references).

The Figure 7.2 overviews the tag propagation. All the tags assigned in part 1 are assigned manually by a user. All the tags in part 2 are inferred hierarchically by either belonging or inheritance in the case of Forms and Reports. All the tags in part 3 are inferred by usage. As the DAO library is catalogued as DataAccess, the usage of the method “OpenRecordset” is also labelled as DataAccess.

Hierarchical tag propagation algorithm. This algorithm tells that the tag of a declared entity is either its particular tag or the tag of its direct container (library, module, etc.). In the case of Forms and Reports, they share the architectural tag of the Form and Report classes available in the library Access. In the case of Tables and Queries, they share the architectural tag of the TableDef and QueryDef classes available in the library Access.

For this, a user can tag a specific library as UI, and all the elements defined within this library are considered UI unless the element is tagged with another tag. This algorithm is used to know the tag of each declaration element. Because it assigns tags based on tree hierarchy, we call this algorithm hierarchical tag propagation.

Usage tag propagation algorithm. Parallely we propose a usage tag propagation algorithm for tagging expressions. This algorithm only tags references: type usage and expressions which refer to a tagged artefact. For doing so, it tells that the tag of an expression is the tag of the referred artefact. Because it assigns tags based on the relation between definitions and usage, we call this algorithm the usage propagation algorithm.

Hierarchically and Usage concerned elements. All the elements in a method have a tag. We call hierarchically concerned elements (HCE) those elements tagged with the same tag as the method containing the code or with the tag *Language*. Figure 7.3 shows a finding of two HCE.

Like this, any usage of, for example, a string library use is considered to be part of the concern that is to be solved by the analysed entity. We call usage-concerned elements (UCE) those that are not hierarchically concerned and calculated through the usage tag propagation algorithm. Figure 7.3 shows a finding of four UCE.

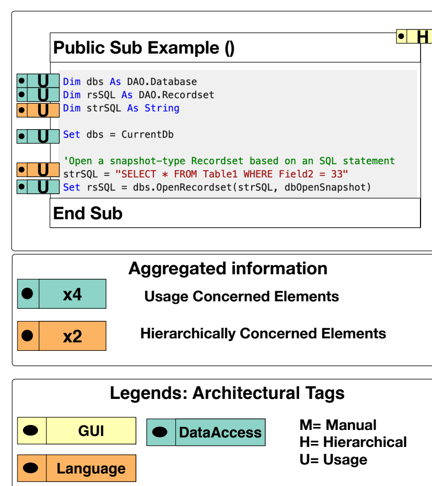


Figure 7.3: Hierarchically and Usage concerned elements (See text)

Summary. After tagging a project, each element has an architectural tag.

This tagging is critical to calculating some of our architectural metrics and visualisations.

7.4 Risks and metrics

The various difficulties related to the migration of Microsoft Access systems (source of the migration) to a web architecture (target of the migration) stem from the incompatibilities between VBA and its paradigm, as well as web applications and their paradigm. In this section, we present a set of metrics that measure these incompatibilities and quantify the gaps between source and destination technologies to measure the inherent risks to the semi-automatic migration process. Each of the following metrics measures the complexity of accomplishing the different requirements discussed in Chapter 3.

7.4.1 Risks related to the relevance of the source code analysis

Often, counting with a parser able to parse all possible programs on a language is key for the automatic and semi-automatic approaches.

Among the set of tools we have developed to work with Microsoft Access projects, we have a VBA parser that takes as input a source code and produces an AST. Our parser is based on the grammar of the VBA language as described in the Microsoft Access documentation. To ensure completeness, we created tests for each grammar case proposed by the documentation. We also test our tool by parsing the Microsoft Northwind Traders [5] example project, covering the full extension of the program.

Risk. Despite our efforts to ensure completeness, we found that at least in one of the companies' projects, our parser fails to produce an AST in %9 of the modules/class modules due to unexpected usage of different grammatical constructions. The lack of documentation coverage of these grammatical formulas threatens the validity of our semantic analysis since we have to interpret what these grammatical composition means, opening the door to ambiguity and misinterpretation. The impossibility of obtaining a model out of code risks any automation, risking all the requirements discussed in Chapter 3.

Parsing error indicator. We use the following Parsing error counting indicator to quantify the parsing errors. In interpreting VBA source code with our parser, we have defined the "SyntaxError" counter. This gives us the amount of parsing errors due to syntax errors (especially those mentioned in the previous paragraph). The higher the value of this counter, the more complex the migration is.

[5] https://docs.microsoft.com/en-us/powerapps/maker/canvas-apps/northwind-install

7.4.2 The risks of language translation

Programming languages often respond to specific formal grammar, which defines a language's limits and possibilities. Many approaches to software migration are based on the interpretation of the semantics of the different source artefacts and the expression of an equivalent semantic in the target language/paradigm. The grammatical constructions or paradigmatic concepts are often incompatible, leading to the inability to express an equivalent semantic.

7.4.2.1 Mapping entities from one formal grammar to another

A critical step in a migration process is the transformation of grammatical entities from the source language to their equivalents in the target language. This consists of associating each source language's grammatical element with one or more elements of the target language. This task is not trivial because it is possible to have elements of the source language with no equivalents in the target language.

Risk. The lack of an equivalent grammatical entity causes a loss of semantics during the transformation. [Terekhov 2000]. VBA is a language that has particularly rich grammar. A given semantics can be expressed in several ways, including information flow control or error handling. Therefore, we find it essential to map the elements of the VBA model that do not have equivalents in the destination environment and to count them when analysing the code of VBA applications. Measuring this risk contributes to understanding the effort required to achieve language transformation: one requirement discussed in Chapter 3.

Incompatible grammatical construct Indicator. Different languages provide different ways to control the flow of execution of a program. Control flow management typically includes conditional branching (if, else if, switch, . . .), loops (such as for, while and repeat, . . .) and error management (such as try/catch,. . .) . There are other less popular and more challenging structures, such as conditional and unconditional jumps (also known as go-to statements). Many VBA grammatical entities are used for error handling and control flow, and for which we do not have an equivalent in Typescript / Java, we mention: Resume Label, Resume Empty, Error Resume Next, OnError GoTo, Resume Next, Property, PropertyAccessors. This grammatical entity indicator counts the number of appearances of these entities in a given source code. The higher the value of this indicator, the more complex the migration project is.

7.4.2.2 Paradigm shift

VBA proposes a hybrid paradigm programming language that aims at developing information technology systems focused on human-machine interaction with a di-

rect impact on an integrated database. We find many concepts that do not exist in the paradigms of our target languages.

Risk. While VBA allows the usage of functions and procedures, all our targets require the code to be expressed in an object-oriented fashion. This requires the identification of concerns and the clusterisation of variables and functions into potential classes. This kind of problem does not have automated solutions since most of the approaches are based on heuristics, and multiple results are possible ([Zou 2001, Martin 2002a, Kontogiannis 1998]). Furthermore, VBA includes many first-class citizens that do not exist in the target environments, such as Tables Queries and Macros. Each entity must be transformed into something else, risking the loss of semantics and consistency. [Terekhov 2000]. Measuring this risk contributes to understanding the effort required to achieve paradigm migration for procedural and access-specific first-class citizens to general-purpose languages: two requirements discussed in Chapter 3.

Indicator for incompatible entities. Counts the apparition of paradigmatic constructions that are unavailable in the destination languages. It estimates the possibility of finding problematic cases during a language translation process. Among all the first-class citizens of the VBA language, we find the followings that have no equivalent on our target platforms: Modules, Tables, Queries, and Macros. The higher the value of this indicator, the more complex the migration project is.

7.4.3 Library migration

A Microsoft Access project, as in most modern languages, relies on using different kinds of libraries. They can be of three types: (i) “BuiltInDependency”: dependencies that are part of Microsoft Access (standard) (ii) “BinaryDependency”: such as DLLs. (iii) “MicrosoftAccessDependency”: dependencies developed in Microsoft Access

None of the libraries is available in the target technologies. All code using these libraries must be migrated. Since the MicrosoftAccessDependency is code that we can migrate, we do not have it into account, while we take into account any “BuiltInDependency” or “BinaryDependency” dependency.

Risk. Many risks are associated with dependencies update, or migration [Cossette 2012]. We consider that there is a high probability that the same library will not be available in different runtime and languages, [Brant 2010]. Therefore, each element defined in a library used by the program must be migrated and required to modify the code using the libraries. Measuring this risk contributes to understanding the efforts required to achieve library and infrastructure migration, two requirements discussed in Chapter 3

Indicator of library complexity. Counts the use of different elements defined in libraries. The indicator counts library elements used by method, class, and project. It estimates the number of places required to change during language translation since none of the libraries used in Microsoft Access is available in any of our targets. The higher the value of this indicator, the more complex the migration project is.

7.4.4 Architectural tangling

Our software migration entails at least splitting a single monolithic stand-alone application into front-end and back-end applications. Splitting a project into pieces is a challenging endeavour, as we must be able to choose which pieces of code are meant to go to the front end and which to the back end, which are required in both targets and which are not required anymore. Split code is *per se* hard; if the code mixes up, many architectural concerns to split code is near to impossible [Brodie 1995].

The term architectural tanling responds to the number of mashed-up architectural concerns in a piece of code; we use it to characterise the complexity of a piece of code.

Risk. The main risk related to architectural tangling is a piece of code is the impossibility of split code, and therefore to conduct the architectural migration of our project. Many authors explain that a clean architecture, aligned with the target architecture, is compulsory for achieving migration [Brodie 1995, Bianchi 2003, Wu 1997]. If the migrating project does not comply with this restriction, refactoring and rearchichetecturising become critical tasks. Measuring the tangling helps to understand the effort required split an application into multiple targets: one of the architectural requirements discussed in Chapter 3.

7.4.4.1 Architectural Tangling Complexity and Architectural Cohesion Indicators

We contribute two indicators: architectural complexity and architectural cohesion.

A method's *architectural tangling complexity indicator* (ATCI Equation 7.1) is the relation between the different architectural concerns included. A method's *architectural cohesion indicator* (ACI Equation 7.2)measures how related it is to the architectural concern where its class is defined.

By behavioural element. These indicators are calculated by function, procedure or method by the following formulas:

$$ATCI = UCE * HCE \tag{7.1}$$

$$ACI = \frac{UCE}{HCE} \tag{7.2}$$

The ACI aims to be zero whenever there is no other architectural concern but the hierarchical concern and is expected to increase proportionally with the more they appear. The higher values highlight the methods that act as architectural concerns bridges or are entirely misplaced. On the other hand, the ATCI can only be zero if a method is empty. It grows faster when more than two orthogonal concerns are mixed.

By structure element. In both cases, the indicators are calculated by module, class or project as the sum of the inner element's indicators. If all the behavioural elements have zero ACTI or ACI, the class or module also is zero.

Use. We use these indicators mainly to sort visualisations presented below in Section 7.5. They can both be used to have a general idea of how complex it would be to split the source system into a front end and a back end. It can also be used to learn which part of the system is more architecturally tangled (high ACTI) or which parts seem more random (low ACI).

7.5 Visualisations

In this section, we present metrics and architectural visualisations.

7.5.1 Metrics visualisations

7.5.1.1 Pareto chart

For visualising complexity metrics, we use the Pareto chart.

According to Kan *et al.* [Kan 2006], the purpose of the Pareto chart is to identify essential individuals in a sample of data. The diagram follows the Pareto principle (also known as the 80-20 principle), an empirical phenomenon found in some fields: about 80% of the effects are the product of 20% of the causes. As shown by Figure 7.4, the Pareto chart consists of two graphs: a histogram of frequencies on the measured variable (grouped by project complexity in our case), where the individual values are represented in descending order by bars, and an accumulation line. In our case, we want to identify the 20% of our sub-projects that cover 80% of the risks. This

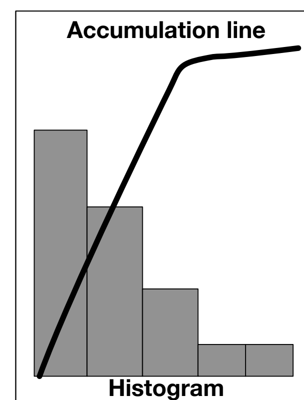


Figure 7.4: Pareto chart schematics.

will help us to focus only on the critical sub-projects. For
this reason, we have decided to use the Pareto chart.

7.5.1.2 Pie chart

Pie charts are used to illustrate the numerical proportion
of each represented category. For the case of grammatical and paradigmatic complexity, we use pie charts to contrast the incompatible entities with the compatible entities. For the case of dependency complexity, we use pie charts to represent the different proportions of used libraries. For the case of architectural tangling complexity, we use pie charts to present the proportions of architectural concerns.

7.5.2 Architectural Analysis and Visualisation

Forms, Reports, Modules and Classes in Microsoft Access can implement multiple concerns such as GUI, GUI navigation, data accessing, interoperability with other systems, and business logic. Knowing that the migration project aims to split the applications into at least front-end and back-end applications, we present a method to recognise architectural concerns and a graph that reveals the concerns of the selected artefact.

7.5.3 Class architectural blueprint

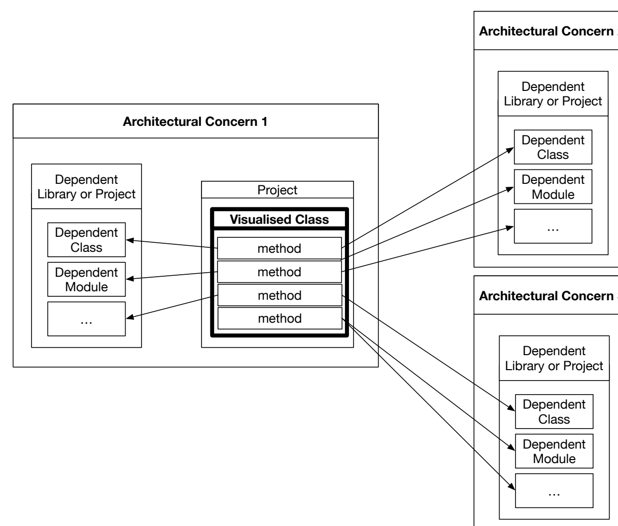


Figure 7.5: Class architectural blueprint schematics. The arrows represent an outgoing dependency of a method. We only show arrows leading to another project or another architectural concern. The visualised classes are clustered by architectural concern

.

For clarity, this chart only renders the uses of components of different projects to offer an overview of the dependencies between all the artefacts within the selected artefact, clusterised by architectural concern.

Figure 7.5 shows a schematic of the class architectural blueprint.

The arrows represent an outgoing dependency of a method. We only show arrows leading to another project or another architectural concern. The visualised classes are clusterised by architectural concern. Only used elements are included in the visualisation, easing the reading.

Refactoring, slicing and joining. Architectural migrations enable an existing system to work in a new architectural environment. New architectural environments often propose different ways to deal with architectural concerns: splitting GUI from business, data access from GUI, etc. The class architectural blueprint is a chart showing which methods in our class rely on other projects and other architectural concerns. This information is valuable to understand, for the given class, what methods should be refactored. In our case, our migration is architectural and language, implying splitting a monolithic application into at least front-end and back-end. For this case, the class architectural blueprint also helps understand what methods will become a remote call.

7.5.4 Architectural tangling highlight

This chart gives an architectural complexity overview of a Microsoft Access project by kind (Form, Report ...) or by entity. Each coloured block represents one behavioural entity (method, function or sub-procedure).

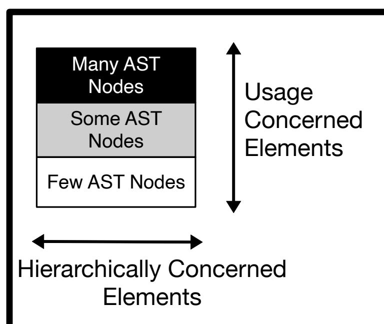


Figure 7.6: Block schematics. (See Text)

- A Block's *width* indicates the number of hierarchically concerned elements within the method.
- A Block's *height* indicates the number of usage concerned elements within the method.
- A Block's *opacity* indicates the number of AST nodes extracted from the element's code. The more elements, the darker the block.

All the dimensions and opacity of the block are on a logarithmic scale. Finally, some blocks are of fixed size and red. These blocks are there to represent external functions and sub-procedures. Figure 7.7 displays the schematics of the chart. Blocks are grouped by ownership: all the blocks representing the behaviour of a

single first-class-citizen (Forms, Modules, Reports or Classes) are together inside a bounding box; these boxes representing first-class-citizen are grouped by kind.

This chart encodes three features of the architectural dependencies per behavioural entity into a block.

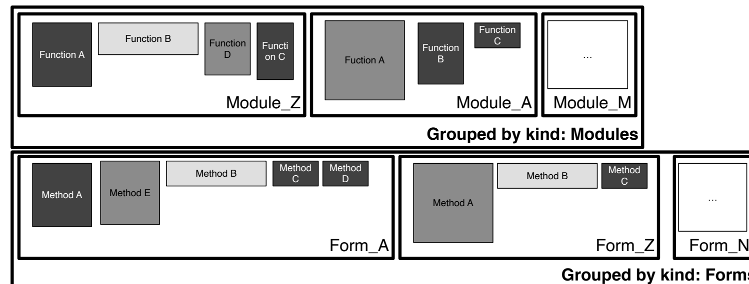


Figure 7.7: Architectural Tangling Highlight schematics: Each block represents a method or function. Blocks are confined within boxes representing their container: class, form, module. Containers are confined in bigger boxes by kind. All is sorted either by ACTI or ACI

.

Blocks are confined in boxes representing the defining entity (class, module, form, etc.). Boxes and Blocks are sorted by cohesion or complexity (explained below in Section 7.4.4). Finally, all the boxes are organised by kind: Table, Query, Module, Report and Form.

Figure 7.8 proposes three stereotypes to ease reading.

- Stereotype 1: The method uses elements that belong to the same and different architectural concerns. It is related to probable architectural layer violation.

- Stereotype 2: The method uses many elements that belong to the same architectures and few that belong to other architectures. It is related to no architectural layer violation.

- Stereotype 3: The method uses many elements that belong to different architectures and a few that belong to the same architectures. It is related to either probable architectural layer violation or misplacement (should this method belong to this class?).

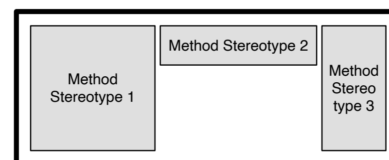


Figure 7.8: Stereotypes for fast reading architectural tangling highlight (See text).

Finding the needles in the haystack. Architectural migrations enable an existing system to work in a new architectural environment. New architectural environments often propose different ways to deal with architectural concerns: splitting GUI from business, data access from GUI, etc. Often, before conducting architectural migrations, an architectural renovation is required. This chart assesses the understanding of *what to renovate*: all elements responding to stereotypes 1 and 3, as they are most likely architectural violations in the target platform. The *how to renovate* is most likely to refactor the elements producing architectural violation into elements of stereotype 2. Elements responding to stereotype 2 are likely not to change. *When to stop*: when few or no element responds to stereotypes 1 and 3.

What needle are we looking for? Having two different metrics to sort is helpful because it puts the focus on the architectural subjects we are looking for. When using ATCI, we are willing to understand where the application's architectural *mess* is. When using ACI, we are willing to understand what should be in here (stereotype 2) and elsewhere (stereotype 3). We can also use ACI to understand where the dotted line is to cut through when splitting an application (specific cases of stereotype 1).

7.5.5 Summary

Our tool allows software migration to reveal (i) grammatical, paradigmatic and dependency complexities; (ii) The aggregation analysis of architectural "mess", with the tangling highlight, and (iii) the analysis centred on first-class-citizen (Class, Module, Form, Reports, . . .). All these are calculated using a Microsoft Access model.

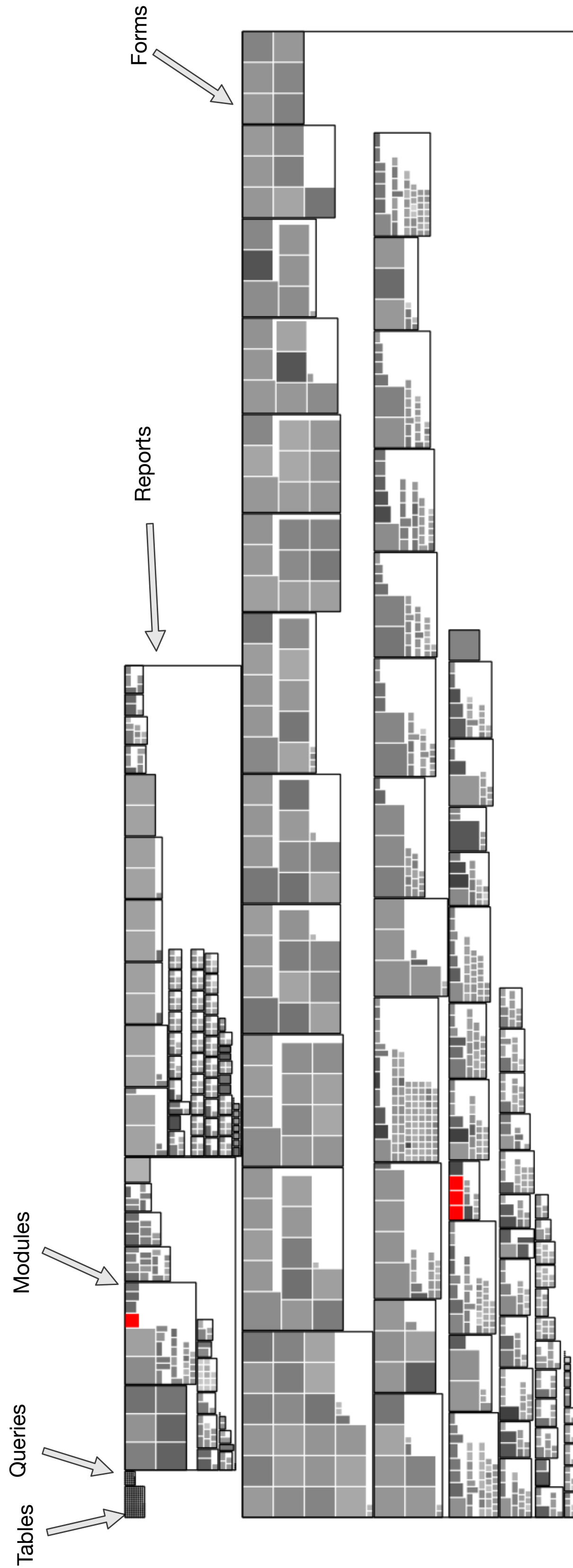


Figure 7.9: Architectural tangling highlight project. This chart highlights all the architecturally relevant elements of the Agent sub-project. The elements are sorted by ATCI and split by kind: Table, Query, Module, Reports, and Forms. This chart does not contain any Class.

7.6 Alce exemplified on the *ePaie* project

One of our real-life experiences in our company is to help key developers in the development of a migration blueprint for the *ePaie* system.

A consultant company with experience in software migration has been hired to issue a migration plan. This company requires the key developers to build a blueprint of the legacy system. The blueprint reports the architectural and functional reality of the migrating project. In this section, we report the process we use to handle relevant information to build such a report. We expose how we use the tool for our case study, focusing on the Agent sub-project.

7.6.1 The ePaie system

ePaie is a Microsoft Access system comprising 18 sub-projects, summing up 1000 UI widgets, 700k lines of code and using 19 different libraries. The target technology of the migration is Java+Springboot for the back end and Typescript+Angular for the front end. *Agent* is one of the 18 different sub-project of *ePaie*. The chapter's examples come from the source of the *Agent* sub-project.

Architectural tangling highlight. The architectural tangling highlight chart gives us an overview of all we have in a sub-project regarding architectural complexity.

Figure 7.9 Highlights all the architecturally relevant elements of the Agent sub-project. The blocks in this chart are sorted by using their ATCI value.

With this chart, we quickly filter out the 12 architecturally most complex forms out of 68 (which is the first line of the forms in the Figure 7.9). We will likely find architectural layer violations and highly complex pieces of code within these 12 forms.

We pick the first form, the most complex in general (even when it does not hold the most complex method). We magnify the architectural tangling highlight in Figure 7.10. Most methods respond to the *stereotype one*. There is one apparition of *stereotype 3*, which means that the method may even be misplaced in this form.

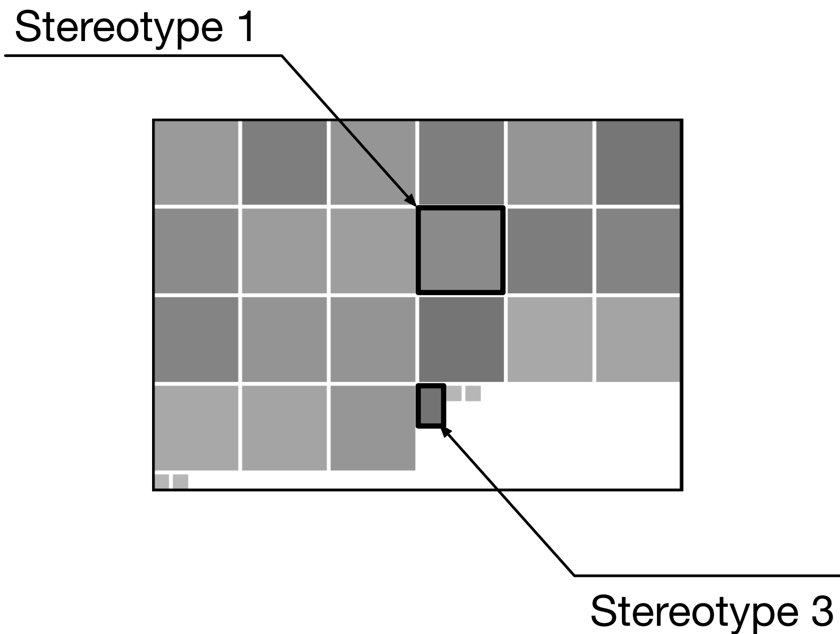


Figure 7.10: Architectural Tangling Highlight: Magnification of the first and more complex Form in the Agent sub-project: Form_FagtVisiteMedicaleSelection.

Dependency pie chart. We can use the dependency metric on its pie chart version to get an overview of the architectural concerns mixed up in our form. Figure 7.11 shows all the different architectural concerns found in the selected form.

With this overview, we can easily know if there is something smelly or not on the implied architectural concerns. This case is special. There are too many concerns. We want to look a bit deeper to understand our dependencies.

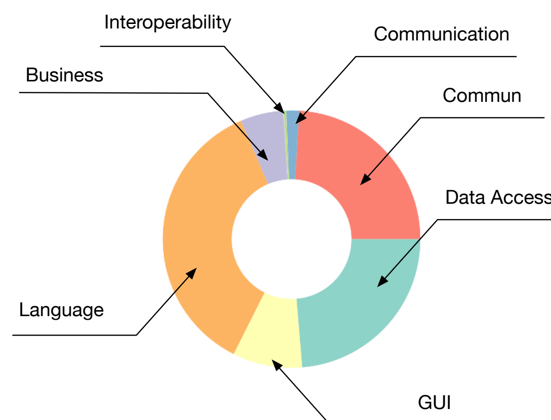


Figure 7.11: Dependency Pie chart: All the architectural concerns found in Form: Form_FagtVisiteMedicaleSelection.

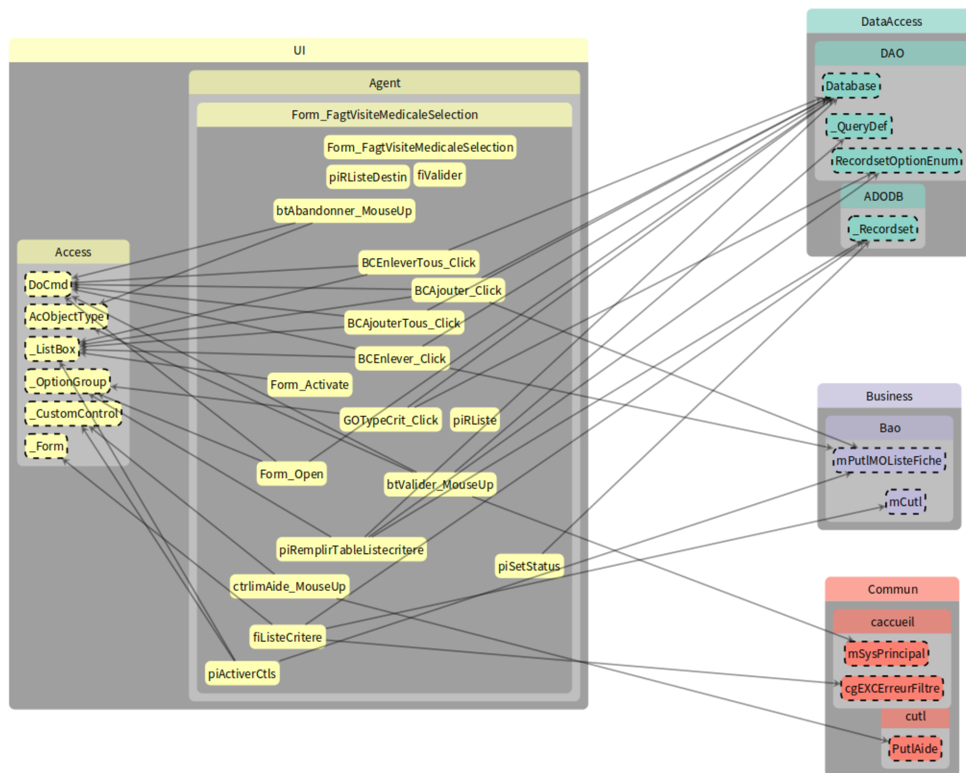


Figure 7.12: Architectural dependency graph applied to Form_FagtVisiteMedicaleSelection. We can observe here a strong dependency on Data Access. Yellow represents GUI. Green represents DataAccess. Purple represents Business, and Red represents Commun.

Class architectural blueprint. In the previous step, we found evidence that this form is concerned with many things simultaneously. To better understand the responsibilities, we use the class architectural blueprint showing us what functions and sub-procedures are using elements belonging to what architectural concern.

The Figure 7.12 shows the dependency graph of this Form. This chart is navigable. This means the user can dive into the details and learn which elements are used in each connection from each architectural concern (functions, types, . . .). This chart also allows collapse to focus only on the architectural connections: in our case, the form relies on Data Access, Business and Commun.

Once we read the usage details, we find that one kind of usage is undesirable: the direct tangling with Data Access features. Even more, this form uses two different ways to access the database: DAO and ADODB libraries.

If we follow the arrows, we can easily differentiate the functions and sub-procedures that do data access from those that do not.

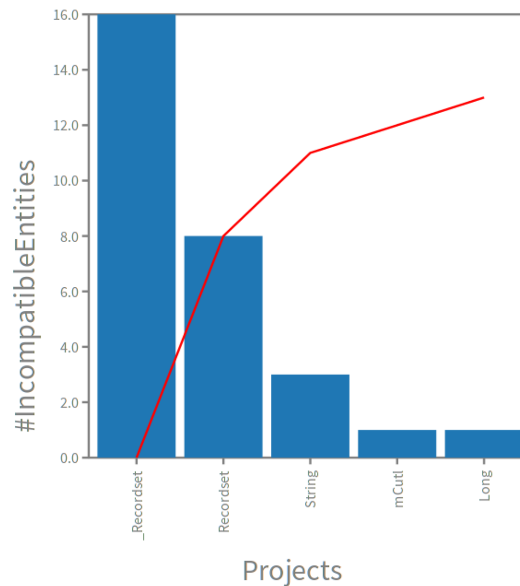


Figure 7.13: Analysing fiListeCritere: Dependency complexity metric on Pareto chart.

Analysing fiListeCritere. We choose fiListeCritere because it is one of the richest methods, using GUI, Commun, Business and DataAccess. It is a function with two parameters. It is written along 48 lines of code without counting comments or empty lines. It parses into 237 AST nodes. Based on the parameters and information obtained from the form, it is a function that builds a query by concatenating strings, directly accesses the database, and generates a string containing information split by a special character. For a first overview of the method, we use the metrics of this specific function, visualising them with the Pareto chart. The grammar complexity chart is not interesting at the function level since it does not reveal much more than the existence of two incompatible entities. The dependency complexity of this function is more revealing, shown in Figure 7.13: the method uses two kinds of recordset from two different libraries to access the database (_Recordset belongs to ADODB, Recordset to DAO). There are 24 uses related to the database.

7.7 Discussion

The metrics we proposed are based on our understanding of the problems encountered in the literature and some found in our migrating experience. Regardless of the link with previous empirical experiments, our work still has room for improvement since all the proposed metrics measure complexity in **Nominal Scale** units. Variability and Reliability have not been tested nor enhanced.

Nominal Scale. Nominal scale metrics are useful to understand how many entities we have in a continuum. This is useful to get an idea of how many of these entities we are bound to find but nothing more. More work is required to establish the contribution of each of these variables to the complexity of the migration.

Validability and Reliability. Validability and Reliability are two fundamental aspects of measurement [Kan 2006], and they must be measured, validated, and empirically proofed. While validability can be enhanced and validated by more exact means, the reliability of the metric requires empirical validation, which implies the requirement of statistical samples on the usage of such metrics.

Uses. Our metrics work has already been leveraged to detect projects and files that are more likely to be interesting for studying layer violations on Microsoft Access elements, building blueprints and understanding the challenges to be faced by an automatic approach to software migration. Still, we need to use all these artefacts in other migrating experiments and survey the usage of these tools by the different stakeholders of software migration. Such surveys are complex to conduct as the required expertise to respond to a survey on the utility of our context is unique.

7.8 Conclusion

Assessing and predicting outcomes of software evolution is quite a challenging domain. We recognise that much work has been done on developing useful tools for assessing our fellow software engineers and other human beings. We developed a tool that puts this technology in the quite specific perspective of software migration because it is the only way we found to make it a tool that is meaningful for our project and our stakeholders.

This chapter presents our software migration analysis tool comprising thoroughly designed metrics and visualisations. Our metrics include the understanding and measuring of (i) Language migration complexity; (ii) Paradigm shift complexity; (iii) Dependencies migration complexity; (iv) Architectural tangling complexity; and (v) Architectural cohesion.

Our visualisations include two designs: (i) Class architectural blueprint; (ii) Architectural tangling complexity highlight. We also use other visualisations to understand the metrics: (i) Pareto chart and (ii) Pie charts.

These elements help us understand the effort required to achieve the different requirements discussed in Chapter 3: Programming language migration, Library and infrastructure migration, and Architectural migration. In Chapter 8, we apply all these metrics and visualisations to real-life scenarios.

CHAPTER 8

Cases of study: Planning and assessing risk and complexity of software migration

Contents

8.1 Risk and complexity assessment for the eGRC project 104

8.2 Risk and complexity assessment for the CyclePaie project 107

8.3 Selecting and prioritising the tasks involving library migration for the *ePaie* project 111

8.4 Conclusion 114

In this chapter, we present three industrial studies where we used our metrics and visualisations presented in Chapter 7.

In Section 8.1, we present a risk and complexity assessment of the migration of the eGRC system; these results were presented in [Bragagnolo 2021a].

In Section 8.2, we present a risk and complexity assessment of the migration of the CyclePaie project. CyclePaie is the most complex project of the ePaie system.

In Section 8.3, we present the process of library migration planning based on selecting prioritising tasks using the Pareto principle over the different used library artefacts.

8.1 Risk and complexity assessment for the eGRC project

8.1.1 eGRC software characteristics

The project eGRC is a highly complex system supporting multiple french public services, from city hall finance to cemetery management. Indeed, it contains all the risks mentioned in the previous sections. Therefore, to get a clear picture of the proportion of these risks in the eGRC project, we have applied all the above metrics in our VBA code parser. To understand these results, we provide graphs that explain the proportion of risks in eGRC and its sub-projects. As a demographic, eGRC counts with *900.000 LoC* distributed in between *1232 widgets*, *564 reports*, *271 function-modules*, *491 class-modules* and *18 macros*. These different modules are implemented by using a total of *21 libraries*, *1172 queries* and *1437 tables*. This project has a heavy load of business rules since it manages several public services such as electoral planning, civil status and cemetery management.

8.1.2 Syntactic errors complexity

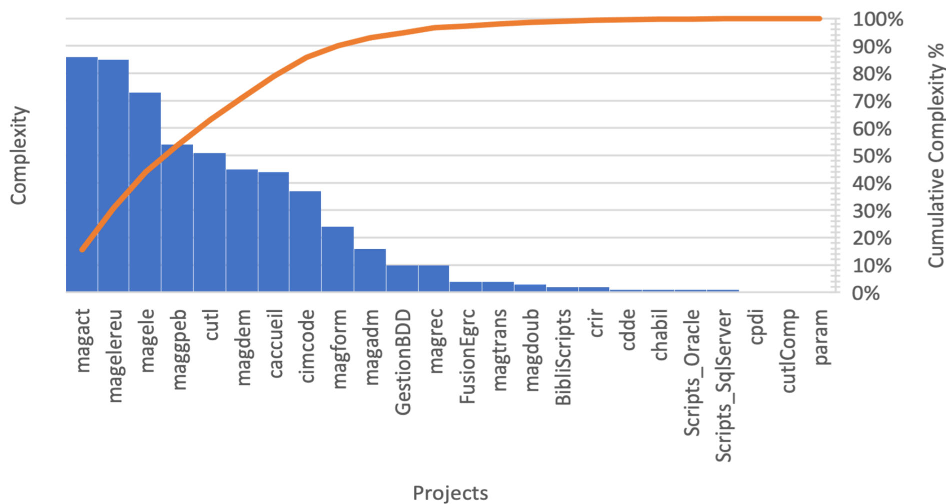


Figure 8.1: Study of the complexity of syntactic errors.

The objective is to show the coverage of our parser by making explicit the grammatical entities not recognised by our code analysis tool. The histogram in Figure 8.1 shows the number of syntactic errors for each eGRC sub-project. Example: The (magact) has just over 85 syntax errors. The accumulation line shows us the percentage of the cumulative frequency. Example: If we solve the syntax

problems in the magact sub-project, we will cover 15% of the total number of syntax errors in the eGRC project, and if we solve the syntax problems in both the magact and magelereu sub-projects, we will cover just over 30% of the total number of syntax errors in the eGRC project. To get 80% coverage, we need to solve the syntax problems in the first seven sub-projects.

8.1.3 Grammatical complexity

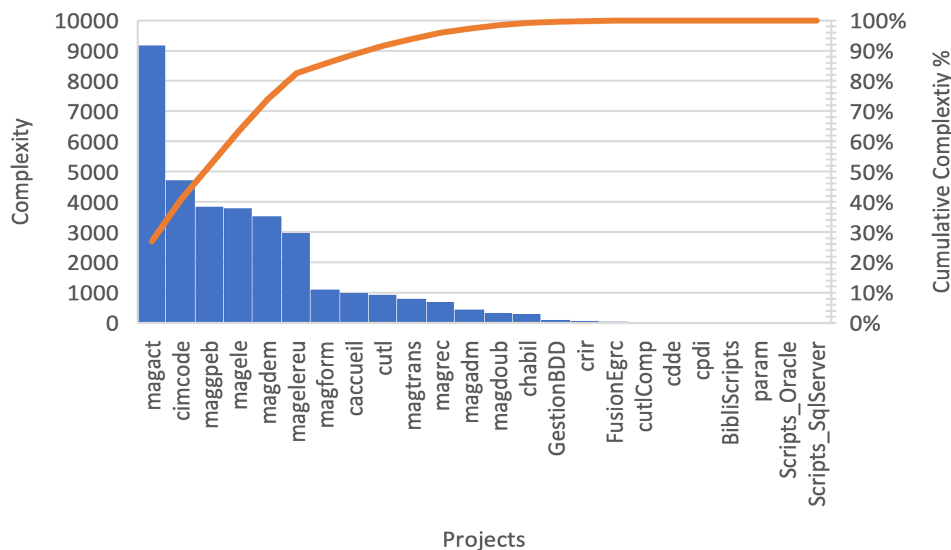


Figure 8.2: Study of the complexity related to the differences in the source.

The objective is to show the degree of mismatch between the grammar of VBA and Typescript/Java. This consists of counting the number of grammatical elements in the source language (VBA) that have no equivalent in the target languages (Typescript/Java). The histogram in Figure 8.2 shows us the number of occurrences of grammatical elements for which we have no equivalent in each of the eGRC sub-projects. Example: The (magact) has just over 9000 occurrences of grammatical elements with no Typescript/Java equivalent. If we solve the equivalence problems in the for it, we will cover 28% of the total number of elements without Typescript/Java equivalents in the eGRC project. To achieve 80% coverage, we need to solve the problems of grammatical elements without equivalents in the first six sub-projects.

8.1.4 Paradigm shift complexity

As mentioned in a previous section, many of the notions related to the hybrid paradigm of VBA have no equivalent in the object and component-oriented paradigm:

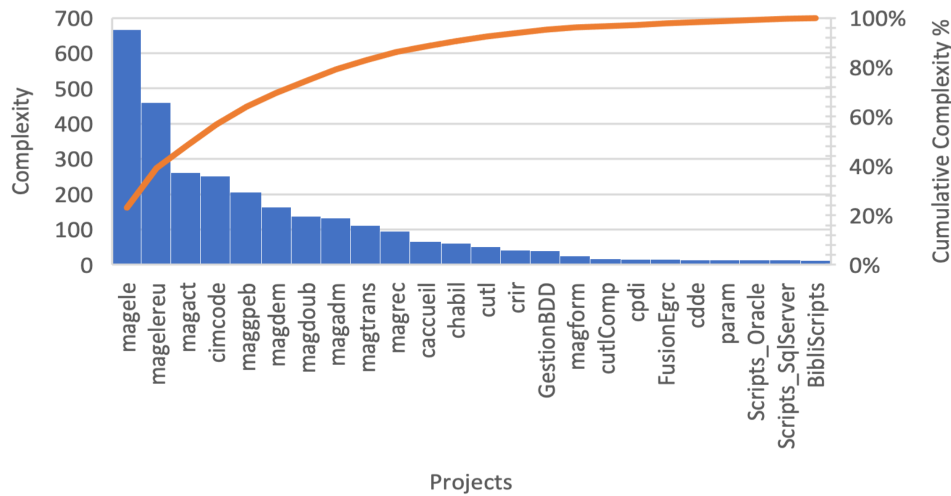


Figure 8.3: Study of the complexity related to the paradigm shift.

Modules, Tables, Queries, Macros, etc. The objective is to show the degree of mismatch between the VBA and Typescript/Java paradigms. This consists of counting the number of paradigm elements in the source language (VBA) that do not have equivalents in the target languages (Typescript/Java). The histogram in Figure 8.3 shows us the number of occurrences of paradigm elements for which we have no equivalents in each eGRC sub-project. Example: the (magact) has just over 650 occurrences of paradigm elements without Typescript/Java equivalents. If we solve the equivalence problems in the magele sub-project, we will cover more than 25% of the total number of elements without Typescript/Java equivalents in the eGRC project. To achieve 80% coverage, we need to solve the problems of paradigm elements without equivalents in the first eight sub-projects.

8.1.5 Dependency use complexity

The objective is to show the degree of dependencies used in each eGRC sub-project. The histogram shows the number of occurrences of dependencies in each of the eGRC sub-projects. Example: The (magact) has 17 occurrences of dependencies. If we solve the dependencies in the magact, magelereu, magform, maggpeg subprojects, we will only cover 30% of the total number of occurrences of dependencies in the eGRC project. All dependencies must be handled in the same way, this can be very time-consuming. To achieve 80% coverage, we need to solve the first 15 projects.

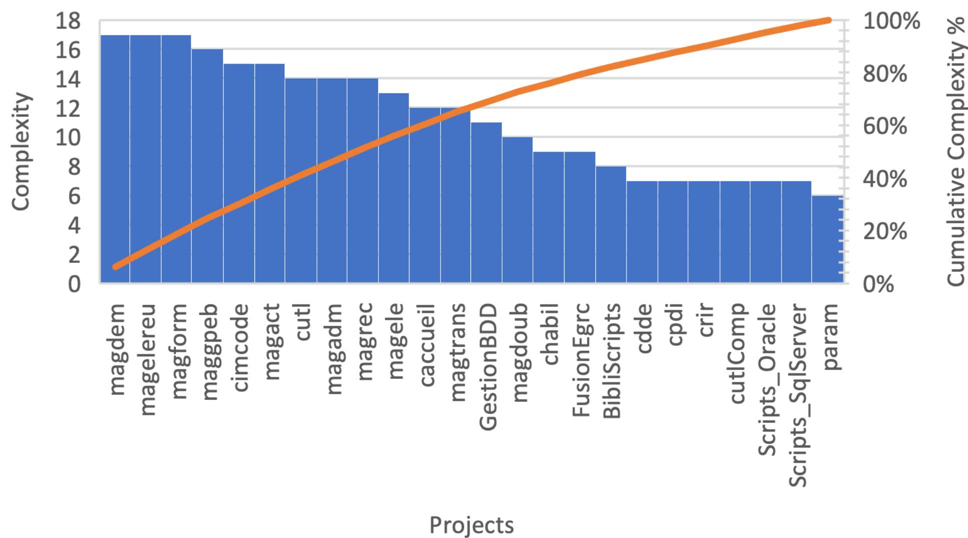


Figure 8.4: Study of the complexity of the use of dependencies.

8.2 Risk and complexity assessment for the CyclePaie project

CyclePaie is the most complex sub-project of ePaie. This section analyses its complexity and all the sub-projects CyclePaie relies on.

8.2.1 Grammatical complexity

The objective is to show the degree of mismatch between the grammar of VBA and Typescript/Java. This consists of counting the source language's grammatical elements (VBA) that do not have equivalents in the target languages (Typescript/Java). The histogram in Figure 8.5 shows us the number of occurrences of grammatical elements for which we have no equivalent in each CyclePaie-related sub-projects.

Example: the CyclePaie sub-project has a little more than 15000 occurrences of grammatical elements without equivalents in Typescript/Java. The accumulation line shows us the percentages of the cumulative frequency. We find a significant accumulation in CyclePaie.

Example: if we solve the equivalence problems in the CyclePaie sub-project, we will cover 80% of the total number of elements without Typescript/Java equivalents in the CyclePaie sub-project.

To have 80% coverage, we need to solve the problems of grammatical elements without equivalents in the first project. We can also see how the consolidation of the rest of the projects does not manage to accumulate more than 20%.

Figure 8.6 shows the pie chart that the ratio of incompatibility is important for

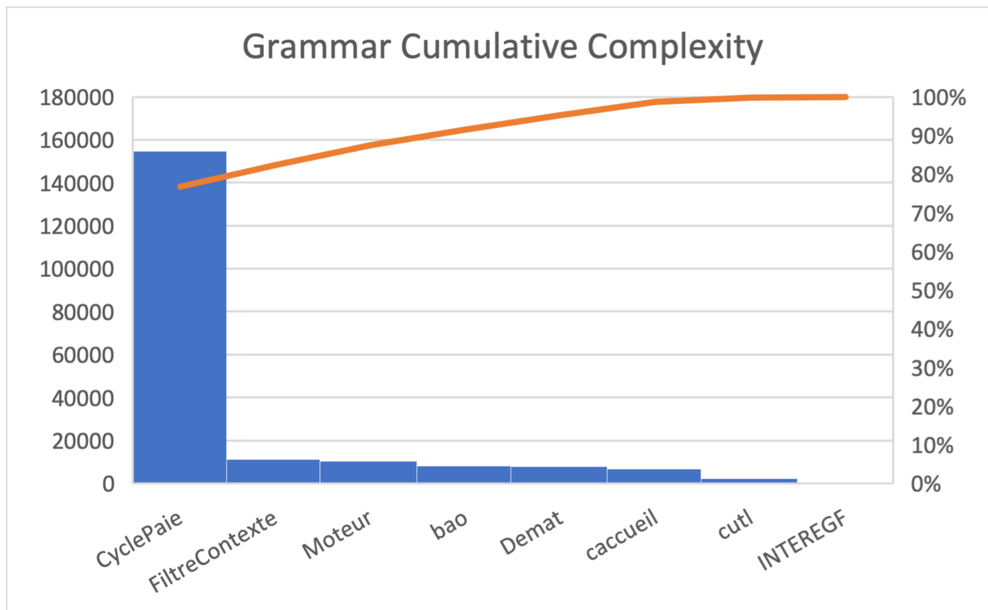


Figure 8.5: Study of the complexity related to grammatical incompatibilities.

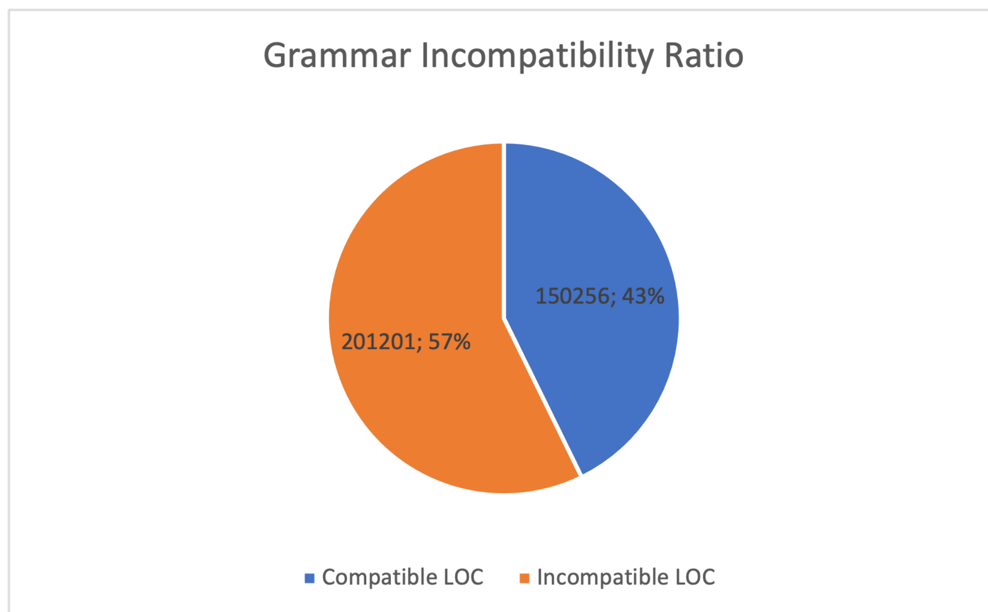


Figure 8.6: Study of the complexity related to grammatical incompatibilities: contrasting compatible and incompatible entities.

the whole project, knowing that it affects 57% of the project.

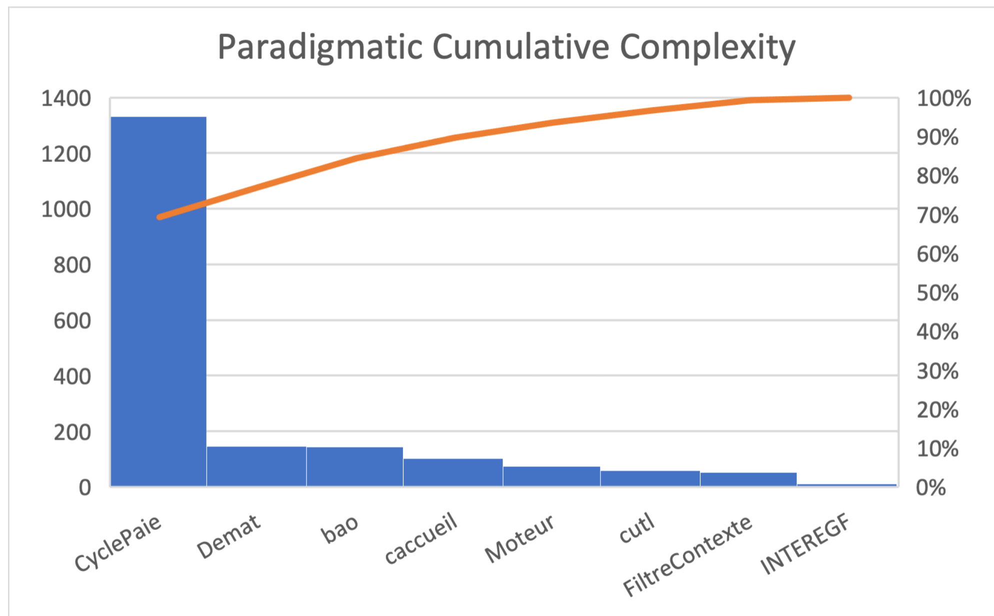


Figure 8.7: Study of the complexity related to paradigm shift incompatibilities.

8.2.2 Paradigm shift complexity

As mentioned in a previous section, many notions related to the hybrid paradigm of VBA have no equivalent in the object and component-oriented paradigm: Modules, Tables, Queries, Macros, etc. The objective is to show the degree of mismatch between the VBA and Typescript/Java paradigms. This consists in counting the number of paradigm elements in the source language (VBA) that do not have equivalents in the target languages (Typescript/Java). The histogram in Figure 8.7 shows us the number of occurrences of paradigm elements for which we have no equivalent in each of the CyclePaie-related sub-projects. Example: the CyclePaie sub-project has a little more than 1300 occurrences of paradigm elements without equivalents in Typescript/Java. The accumulation line shows us the percentages of the cumulative frequency. Example: In this example, we find a large accumulation of paradigm incompatibilities in the CyclePaie project. So much so that it represents almost 70% of the total number of entities to be migrated. This information is critical to understand the need for delegating responsibilities in different sub-projects. It also shows that the decision to represent the different first-class citizens of Access in the destination technology is very important to automate the process. It should be added that mapping the first-class citizen also affects the design and architecture decisions resulting from the migration (especially in the error-handling strategy). However, it is also a critical decision for manual migration.

And also, at the project level, the incompatible entities at the macro level are 93%. This fact confirms the importance of the First class Citizen mapping decisions during the whole project migration.

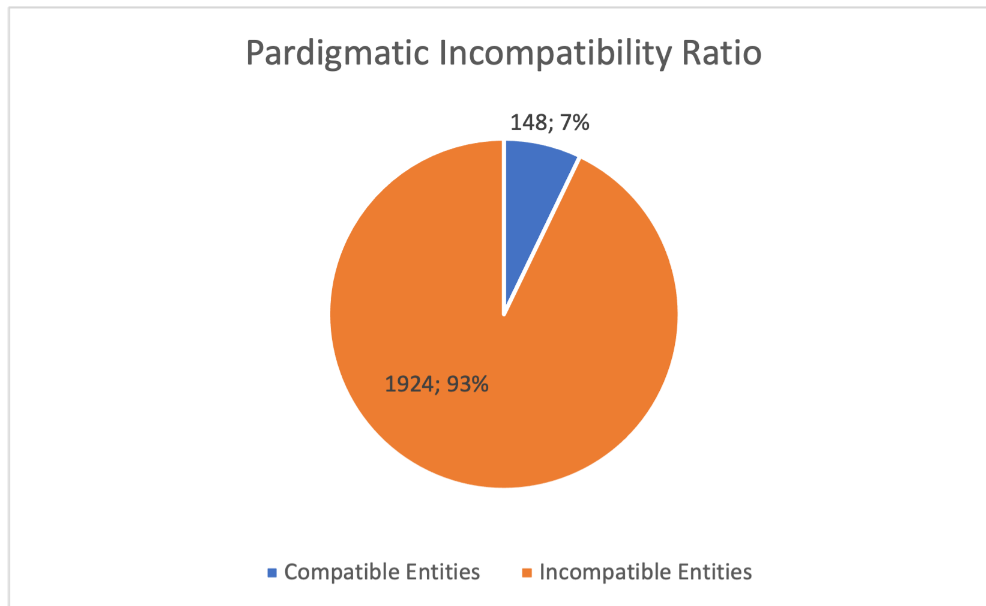


Figure 8.8: Study of the complexity related to grammatical incompatibilities: contrasting compatible and incompatible entities.

8.2.3 Dependency use complexity

The objective is to show the degree of dependencies used in each eGRC sub-project. The histogram shows the number of occurrences of dependencies in each of the eGRC sub-projects.

The objective is to show the degree of use of dependencies in each of the CyclePaie-related sub-projects. The histogram shows the quantity of the different artefacts used defined in other dependencies than that of the same project and language.

Example: the CyclePaie sub-project depends on more than 100 types/functions defined in different dependencies. The accumulation line shows us the percentages of the cumulative frequency. Example: if we solve the problems of dependencies in the first two subprojects, we will cover around 30% of the total number of occurrences of dependencies in the CyclePaie project. We also find that to solve 80% of the dependencies, we have to solve the first 4 to 5 projects.

We find that how to solve the service given by libraries in the destination technology is critical and not simple. We should also add that the impact of these decisions is not measured. Libraries with different APIs will normally have more impact on the code. Finally, in Figure 8.10, we can see that in the context of using library artefacts, those defined by the language have macro levels of 29%. This confirms that deciding what to use as dependencies affects around 71% of the project.

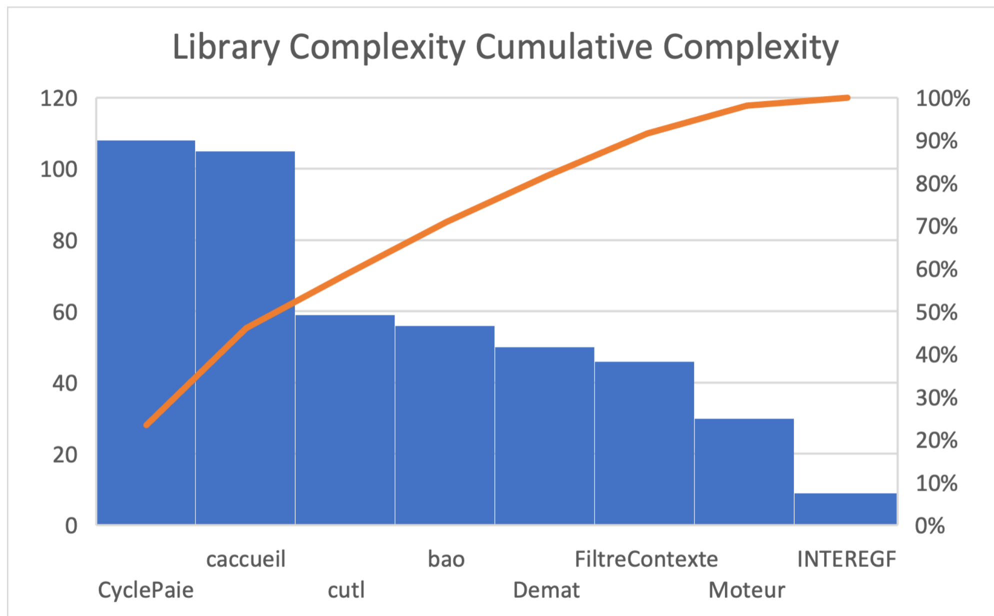


Figure 8.9: Study of the complexity related to dependency usage.

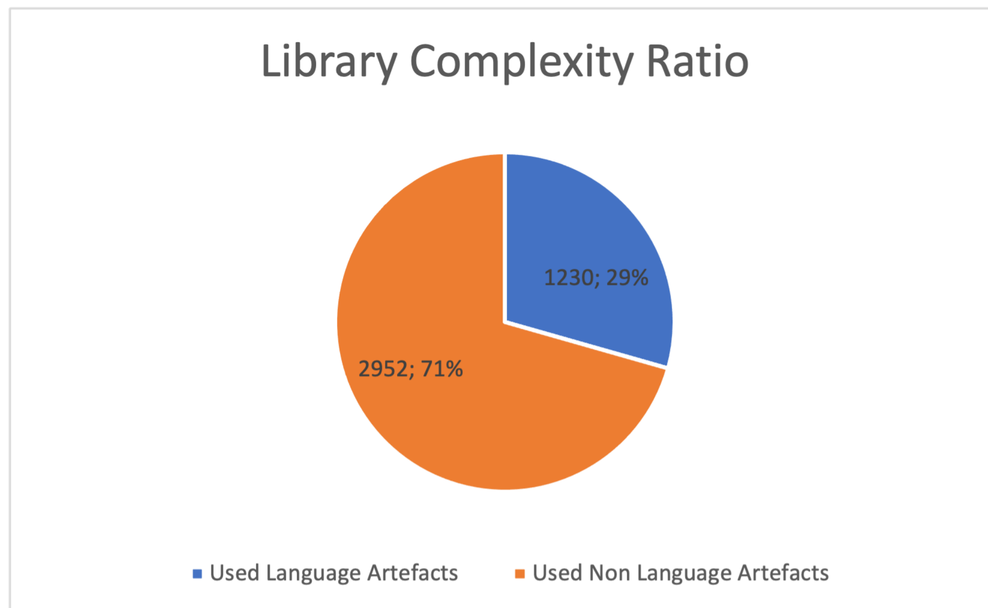


Figure 8.10: Study of the complexity related to dependency usage.

8.3 Selecting and prioritising the tasks involving library migration for the *ePaie* project

This study case arose when we were constructing a Microsoft Access application representing the library usage of one of our industrial projects.

Section 8.3.1 presents the context and need for this study. Section 8.3.2 presents the indicators we used and how we rearranged the aggregation to suit our case. Section 8.3.3 presents a series of Pareto diagrams representing different kinds of artefacts belonging to libraries used by our project. Using the Pareto rule, we choose specific library elements and prioritise them by the number of uses.

8.3.1 The study case

During the migration process, we need to plan how to replace different library elements of the source environment with those in the target. Especially for our case since our migrating approach relies on code translation. Microsoft Access offers no basic libraries available or desirable for our target environments.

All the code we translate must be adapted to the target environment libraries. We have to devise different strategies to adapt the translated source code to use another library in a target environment.

Let's consider the translation to java in the following two examples:

- All the usages of *CStr(x)* (a function that receives any element and returns its string representation) must be replaced with *x.toString()*;

- All the usages of *Recordset* (class used for accessing the database) must be replaced by more than one artefact, or it may require the development of a particular class giving the same service since there is not simple equivalent in Java.

This study case is used to validate library migration in Section 10.3.

Specific demographics. The project *ePaie* uses almost 690 elements from different libraries. To have strategies for 690 different elements is a lot of work. This work requires prioritisation and some criteria to recognise its importance in translation. Instead of rushing to have strategies and implementations for all the 690 cases, we propose using the dependency metrics for the whole project in a Pareto chart.

8.3.2 Information extraction tools

We argue that, in this case, we must distinguish types from functions and variables or constants since each element accomplishes a different task in our code. This is why we refined the dependency metric to be able to measure only one kind of element at a time: (i) function, sub-procedure or method; (ii) member (access to some kind variable: global, class variable, etc) ; (iii) types.

The Pareto chart already tells us which artefacts to take care of first. We propose the usage of this chart to evaluate the most representative set of artefacts, therefore,

which artefacts we must focus on first. And to use the same order to prioritise the tasks.

8.3.3 Results

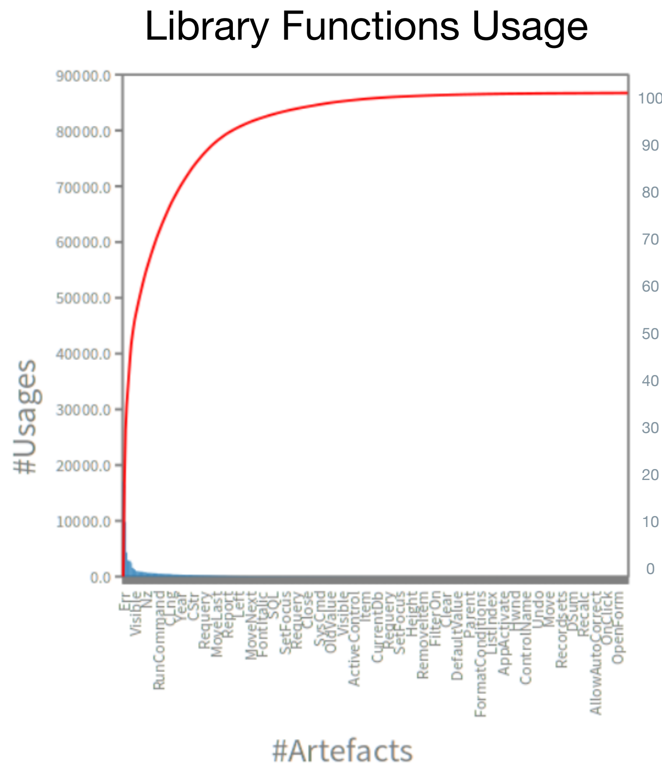


Figure 8.11: Analysing *ePaie*: The most used functions. We have to ensure a java replacement of the first 90 functions.

The distance between each tag on the X-axis for the following charts is 10 elements. The first tag belongs to the first element. Figure 8.11 shows a Pareto chart on the usage of behavioural entities defined in different libraries. 80% of the total invocations are reached at the tenth element (at the function named Report). We prefer to focus on the first 90 out of 430 functions – 20%.

Figure 8.12 shows a Pareto chart on the usage of member entities defined in different libraries. 80% of the total invocations are reached between the third and fourth element of the member named Form. This means we have ensured a java replacement for the first 25 out of 160 members – 15%. Figure 8.13 shows a Pareto chart on the usage of types defined in different libraries. 80% of the total invocations are reached at the fourth element, the member named Form. This means we have ensured a java replacement for the first 30 out of 200 types – 15%.

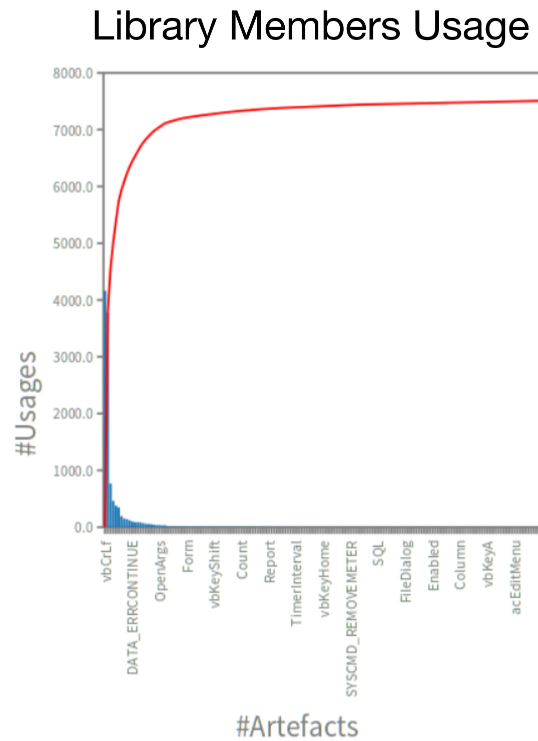


Figure 8.12: Analysing *ePaie*: The most used members (variables, globals, constants). We have to ensure to have a java replacement for the first 25 members.

By applying this method, we can determine the most worthy elements to work with, reducing our initial list of compulsory elements from 690 to 145 – 21% of the total. The rest of the elements can be treated on demand.

We also propose using the Pareto chart's order as the first iteration of task prioritisation: the left-most the first.

8.4 Conclusion

In this chapter, we conduct three studies using the approach proposed in Chapter 7. On the one hand, Section 8.1 and Section 8.2 analysed the risk and complexity of the migration of the eGRC and CyclePaie systems. On the other hand, Section 8.3 presented the process of library migration planning based on selecting prioritising tasks using the Pareto principle over the different used library artefacts.

While the first two reports cannot be yet validated, as the migration of eGRC and CyclePaie has not yet been finished, for what we cannot compare predictions with reality. However, we find that the outcome of the library migration planning presented in Section 8.3 is necessary to devise a representative validation of the transformation approach proposed to achieve migration. We present this transfor-

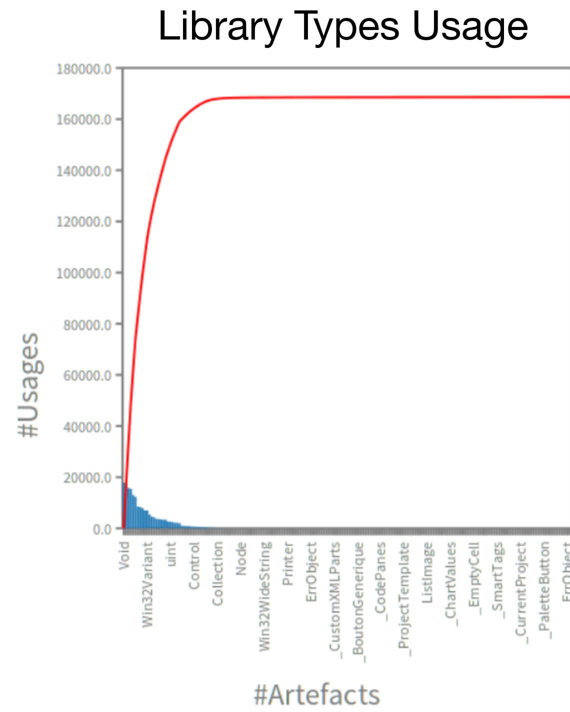


Figure 8.13: Analysing *ePaie*: The most used types. We have to ensure to have a java replacement of the first 30 types.

mation approach in the next Chapter 9.

Part III

Transforming, Modifying and Producing

CHAPTER 9

An Interactive, Iterative, Tooled, Rule-Based Migration

Contents

9.1 Motivations 120
9.2 Overview 120
9.3 Elements of our migration infrastructure 123
9.4 The rule-based context-aware partial migration engine 127
9.5 The approach in action 129
9.6 Discussion 138
9.7 Conclusion 139

In this chapter, we present an approach and a tool to realise an interactive and iterative migration. The chapter is organised as follows: Section 9.2 presents a first high-level view of the approach and its main components. Section 9.3 introduces the different elements of our migration infrastructure, which make our approach possible: models, directives, mappings and rules. Section 9.4 presents the engine: the different challenges and requirements affecting its design and how it puts together the different artefacts to enable software migration. Section 9.5 explains the approach in detail by breaking down into steps the activation of the two proposed directives over a specific case. Section 9.6 discusses different aspects and decisions of our approach.

We note that the content of this chapter is based on our article on submission, "Interactive, Iterative, Tooled, Rule-Based Migration of Microsoft Access to Web Technologies." Bragagnolo *et al.* [Bragagnolo a], and in the technical report "Reporting Context Aware Partial Translation engine based on immediate and delayed Rule application." Bragagnolo *et al.* [Bragagnolo 2022],

9.1 Motivations

The migration constraints are those often encountered in the industry: The software quality is uncertain, development must continue during the migration, resources are limited, the desired target architecture is sketchy, developers are yet to master the target technology fully, and they will be in charge of the evolution of the migrated system in the future.

The last point is considered vital. Despite the changes in architecture and technology, the development team must be able to retain or migrate its knowledge of the application to continue evolving it in the future. To ensure this, we propose that the development team be the driving force behind the migration, choosing what to migrate, where and how to migrate. Yet, as already stated, they are not specialists in the target technologies (Typescript and Angular) and software migration. We propose an interactive, iterative/incremental and tooled migration process to allow them to achieve this migration.

Abstraction. A tooled approach allows software engineers to concentrate on what they want to achieve without considering how to achieve it. For example, they can transform a function into a method without worrying about transformation rules or the syntax of methods in the target language.

Low barrier. An interactive approach enables developers to point to a software artefact, *e.g.* a function, and decide where to migrate it, *e.g.* a class, without worrying *how to* do it. The fact that the approach is based on rules reduces the need for migration experts, as experts have to put their knowledge into reusable fine-tuned rules.

Gradual learning. An iterative and incremental approach allows developers to migrate screens or functionalities one by one. This, plus the feedback provided by a tooled approach, allows one to gradually get acquainted with the migrated solution, its new architecture, and its new technology.

9.2 Overview

We introduce the challenges and the expectations of our migration in Section 9.2.1, and we sketch the process proposed and its different parts in Section 9.2.2.

9.2.1 Requirements

In Chapter 3, we discussed the challenges and requirements of the migration technology and of the migration process.

However, we brief them again to help to understand the different decisions made in our transformation approach.

Technology requirements. The approach must (i) be *agnostic*; (ii) Be capable to *transform* the source code into the expected targets and resolving the need for explicit dependencies through the usage of *import*; (iii) Provide mechanisms to *replace* source *infrastructure* and *libraries* with the target equivalents; (iv) Provide mechanisms to *migrate procedural and MS Access first-class citizens* to object-oriented programming; (v) Provide mechanisms to migrate the architecture by producing *partial* migrations, *split* software into multiple applications and establish *communication* in between these parts; (vi) Provide mechanisms to the *visual* and *behavioural* aspects of the GUI; (vii) Consider the *expertise* of the source application developers and the *etiquette* of the target language to favour the production of code that both developers of the source applications and developers of the target technology can understand.

Migration process requirements. To favour an *agile, Iterative and Incremental* process, we must ensure an approach that is *plan-oriented* and which enables the migration of *core features first*, of *arbitrary-size*. To favour the learning process of an *agile* process, we must provide *immediate feedback* and give the developers the *right to be wrong* and be able to undo tasks. Finally, the process must consider that both the *source* and *target* applications are in *permanent evolution*.

9.2.2 An Iterative & Interactive Process for software migration

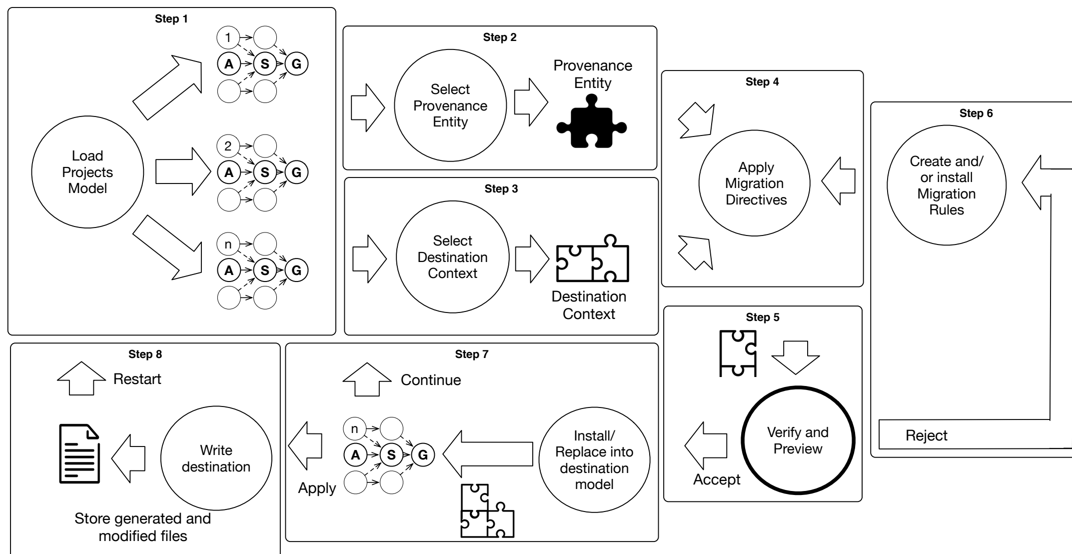


Figure 9.1: Migration Process

The process that we propose is depicted in Figure 9.1; it has the following steps:

Step 1. The migration tool loads the source and target applications and creates a model for each of them. These models are presented to the software developers in the tool as a tree of software artefacts and matching source code (see, for example, Figure 9.2).

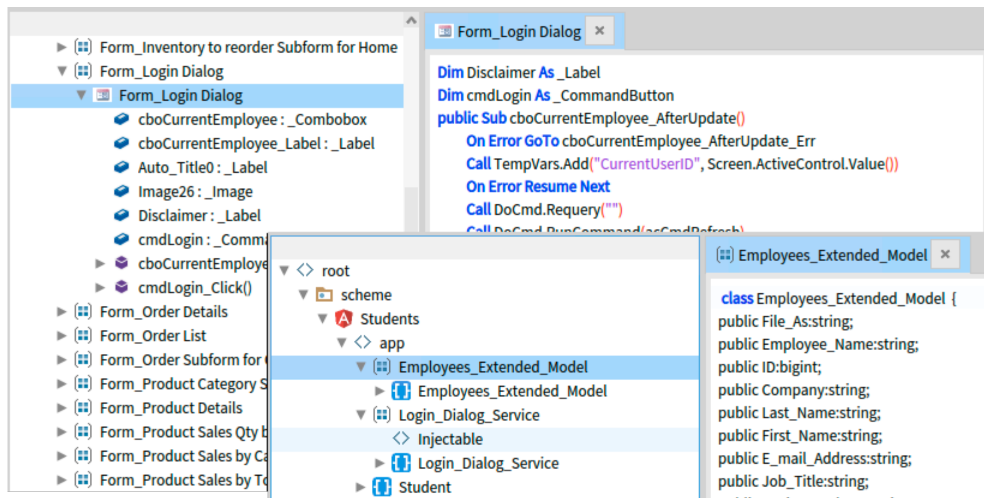


Figure 9.2: Microsoft Access model (top left) and Typescript Angular model (bottom right) presented to the user

Step 2. The user navigates the source hierarchy and selects an artefact to migrate. It could be a module, a form, a function,. . .

Step 3. The user manually chooses where in the target hierarchy he wants to migrate the artefact. The tool provides migration actions for the user to perform. These actions are named *directives* and introduced in Section 9.3.2. The tool accepts any location; however, to be meaningful, it must be coherent with the artefact to migrate, *i.e.* a function can be migrated into a class and become a method.

Step 4. The tool responds to the migration actions by applying migration rules (transformations) to perform the migration. This mechanism is explained in Section 9.5.2 and Section 9.5.3.

Step 5. The user has immediate feedback. If the tool can apply the directive successfully, the user can immediately analyse and verify the code in the targeted model. If the tool fails, the user is informed, and the tool rolls back any intermediate change initiated by the failing directive.

Step 6. If the user is unsatisfied with the result after verifying the code, he can refuse it by rolling back to the previous state. Whatever the reason for a rollback (error or user choice), the migration expert (typically not the application developers) will need to add or change a rule for the process to restart.

Step 7. If the change is accepted, it is introduced in the target model. The user may continue applying directives, going back to step 2 or generating the code.

Step 8. The user chooses to apply all the modifications to the project's source code. This modifies the project files to comply with the new model. After this step, we restart the whole process from step 1.

The tool allows executing the model modifications over the different application's source code at any moment, applying manual changes over any of the applications, and reloading the models.

9.3 Elements of our migration infrastructure

Our approach enables developers to execute arbitrary partial migrations between two different application models (see Section 9.3.1) through the interactive application of *directives* (see Section 9.3.2), which are resolved by the immediate and delayed application of rules (see Section 9.3.4).

9.3.1 Modeling source and target applications

Our model-driven approach applies modifications to target models based on entities and knowledge extracted from source models. All models are instantiated from the same heterogeneous unified meta-model.

To achieve that, we load a model for each application in the migration project. In our primary study case, we target a client/server architecture. We instantiate three different models, one per application: (i) legacy monolithic application (Microsoft Access), (ii) migrated back-end application (Java Springboot) and (iii) migrated front-end application (Typescript Angular).

The Heterogenous Unified Meta-model. Our approach leverages the heterogeneous unified meta-model presented in Chapter 6. This meta-model consists of an Abstract Semantic Graph (ASG) that uses the same entities for different languages as long as the instances have the same semantics: the modelling rationale behind it is to model the likeness and unlikeness of the different constructs. When this is not the case (*e.g.* Java and TypeScript classes), we represent them with different types of nodes to differentiate them in the migration rules.

Representing *inconsistent* intermediate states. Our approach also requires representing *inconsistent* intermediate states as it is iterative and interactive. The migration engineer must be able to (i) write a migration transformation and see the outcome, whether it is or is not correct. (ii) write migration transformations that produce partially correct outcomes due to insufficient information. *e.g.* Let us consider that we have to migrate a function invocation. Suppose we do not know

how the invoked function will be migrated. In that case, we cannot decide how to translate the invocations: method invocation, static method invocation, or function invocation.

Representing *Primitive* and *third party types* and *behaviours*. Modelling of an application in a given programming language also includes creating declaration instances for the programming language's primitive types and any external library functions used. These declarations assert the existence of the library entities and have no complete definition; they are used to be referred by the parts of the application that require them.

9.3.2 Directives: User explicit actions

Our engine exposes two main actions to the user: *Produce* and *Map*. These actions establish a relationship between two entities belonging to two different models. Within the duration of the action, one entity plays the role of the source, and the other plays the role of the target. To convey the direction of the action, we call them *Directives*.

Produce: Given a *source entity* and a *target context*, this directive instructs to produce the given *target entity* inside the given *target context*, based on the *source entity*.

Map: Given a *source declaration*, a *target declaration*, and a *scope of validity* (in the target model), this directive establishes a semantic equivalence between the two declarations, meaning that —within the given scope— past and future references to the source declaration will be replaced by references to the target declaration.

Directives are implemented by migration rules and mappings, as shown in the following sections. These two directives offer the user the activation of two migrating actions.

Actions.

1. the *declaration* of a software entity in a model (*e.g.* a function in the legacy application) must be created as the *declaration* of another software artefact in another model (*e.g.* a method in the migrated application);

2. *references* to a software artefact (*e.g.* a primitive type or a library function in the legacy application) must be replaced by *references* to another equivalent software artefact (*e.g.* a matching primitive type or a replacing library function in the migrated application).

Note that the first action (re-declaration of a software artefact) also entails the second, as the previous references to the original artefact must be changed to references to the migrated one.

9.3.3 Cross-model Mappings & Stubs

Mappings. Mappings are *scoped* objects representing a *semantic equivalence* between two models: a source entity is equivalent to a target entity. *e.g.* (long => bigint); (MsgBox => alert); etc.

By *scoped*, we mean that they have a scope of validity. This is to say that A is equivalent to B for a specific project, package, class, method, etc. This context is defined when recording these mappings. When the mapping is recorded automatically, this scope is the whole project. When the mapping is recorded manually, the user defines its scope. We distinguish two kinds of mappings: Simple and nested.

Simple mapping. This mapping relates a source declaration with a target declaration. It works as a simple association, implying that the source declaration is equivalent to the target declaration in the target ASG. This mapping is enough to map two declarations without parameters or two declarations where one is the migration of the other. *e.g.* (Void => void); (String => String); etc.

Nested mapping. This mapping associates a source declaration with a target declaration and the parameters between source and target declarations. When mapping a function to a method, this mapping allows designing an argument to be the receiver of the method invocation. For example, let's consider the mapping between function F(x,y,z) and method M(a,b,c). Three mappings examples could be:

- (F => M (a => z; b => y; c => x)): all the target parameters are mapped to all source parameters. The order changes.

- (F => M (a => x; b => y; c => x)): parameters a and c are mapped to x; parameter b is mapped to a. Parameter z is dismissed.

- (F => M (a => x; b => y; c => x; z => R)): parameters a and c are mapped to x; parameter b is mapped to a. Parameter z is proposed as the receiver.

Please note that the argument mapping is inversed, as we need to ensure that all the target parameters are assigned to something, regardless of whether there is an equivalent source parameter.

Adaptive rules use mappings (See Section 9.3.4.2). In our approach, a mapping association is a result of a map directive (explained in Section 9.3.3) or the execution of a Productive Rule, as explained in Section 9.3.4.1.

Stubs. A reference node can only point to a declaration node within the same model. When a reference in a model (*e.g.* a migrated application model) should point to a declaration that does not exist in this model (typically a declaration that has not been migrated yet), we create a *stub declaration* in the same model as the reference. The target reference then points to this stub, which points to the source reference used to create this target reference. Therefore *stubs* are bridges between models. As long as it contains stubs, a model cannot generate valid source code because some references still need to be migrated. A specific mechanism replaces these stubs with actual declaration nodes when the latter are created (Section 9.5.2).

9.3.4 Rules

Rules are *scoped* conditional operations consisting of a Condition and an Operation. *Condition* consists of a predicate that satisfies the operation's assumptions and requirements. *Operation* consists of any systematic modification over the target. By *scoped*, we mean that they have a scope of validity. This scope is all migration models, as there are default rules, a specific project, package, class, or method. This context is defined when installing the rules. As rules are objects, they can have their internal state and be configured to specialise their behaviour while creating the rule object.

We have two families or rules: Productive and Adaptive.

9.3.4.1 Productive Rules

These rules create a migrated version of a source entity in the context of a target entity, that is to say, as a child of this target entity. Applying a Productive Rule also entails automatically mapping the source entity and the produced target entity.

Condition consists of a predicate that tells if the rule can produce this migrated version within the specific target entity.

Operation consists of modifying the target entity by defining the source entity's migrated version. These rules are typically applied from the legacy application model —containing the source entity— to a migrated application model —containing the target entity.

For example, a rule is executed when the source entity is a function, and the target context is a class, and it operates a migration by defining a method in the target class based on the source function.

We use productive rules to generate any entity in the target context. Within the rules, we scope different approaches to migration.

Like this, we offer some explicit language translation rules and other rules based on knowledge models. In this chapter, we use only rules that help the translation of language. However, the complexity within a rule is arbitrary and depends on each case of migration. In Chapter 10, we introduce rules using knowledge models as the case study requires.

9.3.4.2 Adaptive rules

We said before that *references* to a software artefact in the source model must be replaced by *references* to another equivalent software artefact in the target model. To enable this replacement mechanism, we copy the *references* as they are, knowing that this may be a wrong decision, and delay the migration criteria to when this other software artefact is available.

Whenever a new mapping is available, the engine checks for adaptive rules, which can leverage this new mapping to detect the apparition of semantically equivalent declarations and modify or replace reference objects based on the nature of the software artefact.

Condition consists of a predicate that tells if the rule can modify or replace a target reference to refer to a given mapping between source and target declaration.

Operation consists of the adaptation of the reference object.

For example, let us consider a function that just migrated as a static method. As soon as the static method is created, the engine maps the function with the static method and activates the adaptive rules on the references that may be affected by this new mapping. Through the application of adaptive rules, the engine will detect that this static method is recognised as equivalent to the function and therefore detect all the uses (migrated before and after the migration of the function) and adapt them from the form `function()` to the form `ClassName.function()`.

9.4 The rule-based context-aware partial migration engine

At the heart of our migrating approach is the *engine* controlling the application of the directives, the selection of the rules and their application. This migration engine has to face the challenge of allowing partial migrations and responding to the user designs before any technologically convenient decision to ensure automation.

9.4.1 Engine's challenges

Software migration is a complex and risky endeavour. As we discussed in Chapter 3, our process is expected to be agile and to allow developers to conduct coarse-grain tasks.

We aim to enable a software migration approach that offers the most needed features and supports a user-directed interactive migration. This is why the engine must enable the late definition of **what**, **how**, and **when** to migrate, as well as ensuring **decision consistency**.

What to migrate. Much of our code may make no sense on the target (*e.g.* technologically explicit code aiming to solve desktop-specific problems.). A target

library, a service, etc., may replace part of the code. Besides allowing the user to decide what to translate, the key to the approach is to subordinate the approach to a human-made plan. For this, the engine must be able to produce partial migrations.

How to migrate. An artefact in a source language may have one or many possible representations in different target languages. For this, the engine must support different strategies to migrate different pieces into different contexts: It is not the same to migrate a widget from Microsoft Access to the front or to the back end. Therefore, the engine must be context-aware. This is why rules and mappings have specific application contexts.

When to migrate. From the process point of view, we do not want to wait to fully finish the migration to deliver some features of the target application. This is why we must subordinate the approach to the user and, by extension, to the migration plan; we must yield control of the migration order. The engine must allow successive partial migrations and be able to articulate them incrementally automatically, producing a similar result, regardless of the order of migration.

Decision consistency. Let's say we migrate a function as a method; for this decision to be consistent, it should impact not only this single artefact but the migration of all the function calling to the migrated function, as they should be migrated as method invocations. As an extension of how to migrate, we have to be able to decide whether to migrate a piece of code as library usage or as an artefact with a different nature. In each case, the decision of *how to migrate* an artefact must be consistent by impacting the migration of all the uses of this artefact.

9.4.2 Executing directives with rules look-up and default behaviour

We need directives and clear interaction with rules to subordinate the migration to the user intentions. In our approach, rules and mappings are contextualised, meaning they only affect a part. We can install a rule or a mapping; thus, it is always applicable or only applicable in the context of specific construction (project, package, class, etc.).

9.4.2.1 Executing the produce directive

When the user initiates a produce directive on a source entity and a target context, the engine searches in the target context for a *Productive Rule* that can handle the migrating case: a rule with a condition predicate that accepts the source entity and target context. The rule has the responsibility of returning the created entity. Whenever the source entity is a reference, the engine traces the source reference

with the rule applied and the created entity. As we explained in Section 9.3.4, rules are contextualised, so we look up rules by context. The search starts in the target context and goes up the ASG as needed. We present in detail its application in Section 9.5.2.

Default Productive Rule: AnyCopy. The produce directive is a direct order of transformation from the user. Therefore, it must be accomplished immediately, so we must have a default rule. As we explained in Section 9.3.1, we base our approach on a heterogenous unified meta-model, meaning that we use the same meta-model for all projects. This choice is no accident. We are required to provide rules only for the cases that require a transformation. For the rest, we provide the AnyCopy Productive Rule.

AnyCopy.

Context: Root context (available regardless of which project is the target)

Condition:

1. Allways true

Operation:

1. Make an instance of the same class as the source entity within the target context.
2. Migrate each of the children of the source entity using the freshly created instance as the target context.

9.4.2.2 Executing the Map directive

When a map directive is initiated by the user, on a source and target declarations, within a context, the engine installs a mapping in the given target context (for example, function F is equivalent to method M only in the package P) right after; the engine searches all the references within the given context to be fixed. Each of them searches for any *adaptive rules* that can be applied after the mapping has been registered, starting with the reference-to-fix context and going up the ASG as needed. Adaptive rules are executed only when there is a mapping available. Therefore there is no default rule. We present in detail its application in Section 9.5.3.

9.5 The approach in action

We illustrate the process of the migration of the VBA sub-procedure “showName” (Listing 9.1), to the context of a Java class named “MyDestination” in the back end

application (Listing 9.2). The source sub-procedure is a simple piece of Visual Basic that pops up a dialogue showing the content of a variable name, concatenated with the string literal "Ms ". Because this is migrated to the back-end application, the migrated class cannot access a GUI. Any information that was previously displayed should now be logged.

```
1 Dim name as String
2 public Sub showName()
3   Call MsgBox ("Ms " & name)
4 End Sub
```

Listing 9.1: VBA sub-procedure that pops up a dialog

```
1 package MyPackage;
2 class MyDestination {
3   public static void log (String) {
4     ...
5   }
6 }
```

Listing 9.2: MyService Java class

9.5.1 Engine setup

We consider the engine configured with specific rules and no previously existing mapping for this example. We now present the installed productive rules AnyCopy, CopyAsStaticMethod and CopyReplaceOperator and the adaptive rule RenameAdaptToStaticReceiver.

9.5.1.1 Productive Rules

AnyCopy. was presented in Section 9.4.2.1.

CopyAsStaticMethod.

Context: Java migrated application

Condition:

1. The source is a behavioural entity (function, subprocedure, ...) **AND**
2. The target context is a class

Operation:

1. Define in the target entity a static method with the same selector as the source entity. If the source entity is a subprocedure, the method is set to use void as returning type reference.
2. Migrate all the children of the source entity using the method as the target context.

CopyReplaceOperator. This rule requires two parameters when it is instantiated: The operator to detect (OtD) and the operator to replace it (OtR).

Context: Java migrated application

Condition:

1. The source entity is a binary operation
2. The operator matches OtD.

Operation:

1. Define a binary operation in the target entity, using OtR as an operator.
2. Migrate all the children of the source entity using the freshly created binary operation as the target context.

9.5.1.2 Adaptive Rule

We use one adaptive rule: RenameAdaptToStaticReceiver.

RenameAdaptToStaticReceiver – Replace by method invocation with *static* receiver .

Context: Java migrated application

Condition:

1. The target reference is a function invocation **AND**
2. The given map target declaration is a method **AND**
3. The given map target declaration is *static*.

Operation:

1. Define a method invocation expression (a reference) using the method's parent as a receiver.
2. Set the arguments used by the replaced invocation into the freshly created method invocation according to the map definition.
3. Configure the freshly created method invocation to refer to the mapped target method declaration.

9.5.1.3 Rule installation

We must install the rules we want for our example to complete the engine setup. As we said in Section 9.3.4, rules are *contextualized*. As we said in Section 9.4.2 the engine performs a lookup through the target model to find rules and mappings. Figure 9.3 depicts our example installation. AnyCopy is always available regardless of the targeted project. In the context of the target project, we find the rules defined above: CopyAsStaticMethod, CopyReplaceOperator and RenameAdaptToStaticReceiver.

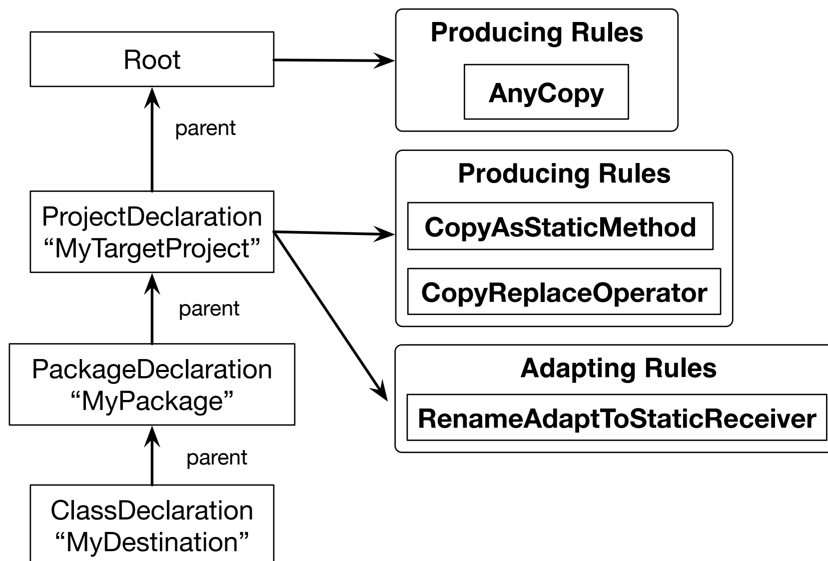


Figure 9.3: Rules by target context. On the left side, we find the top levels of the hierarchy of the target ASG. On the right side, we find the installed Productive and Adaptive rules.

9.5.2 Example of the application of a produce directive

The produce directive starts when a user asks to migrate the specific source declaration, the sub-procedure "showName", into a specific target declaration, "MyDestination" class. The result of the application of the produce directive is displayed in Figure 9.4 and can be used to follow the migration. We break down the example into eight steps. Figure 9.4 shows the source model and the target model after the execution of this directive. Each number attached to a target model entity is the step where they are created.

Step 1 The engine searches for a *Productive Rule* in the context of the "MyDestination" class (of the target model) and does not find it; next, it looks in its

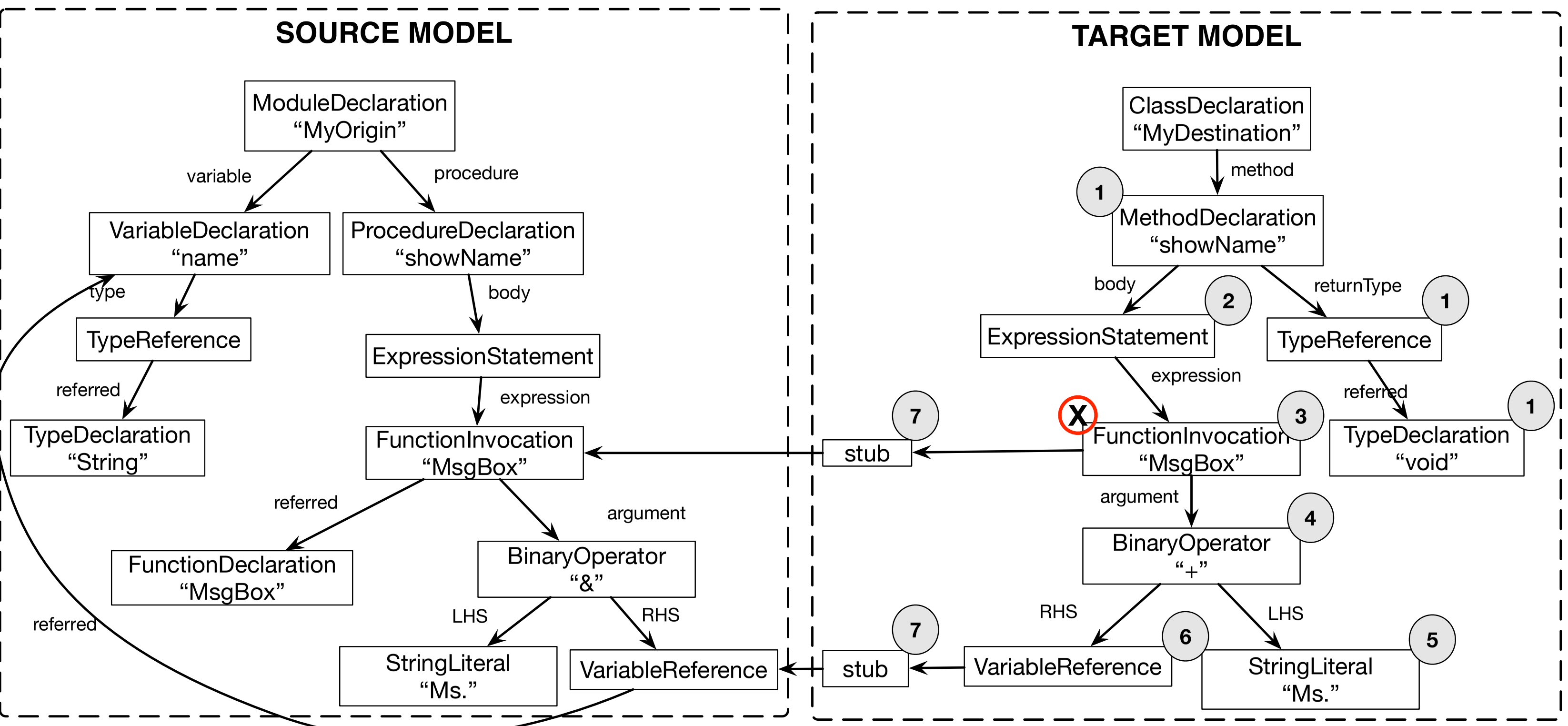


Figure 9.4: Produced “showName” method from “showName” sub-procedure. BinaryOperator and StringLiteral are grammatical nodes; the FunctionInvocation node is not valid in the target model. The numbers in the visualisation refer to the step when the different elements have been created. We note the difference between children and references.

parent context and up in the context hierarchy until the top level node of the Java migrated application model where the CopyAsStaticMethod rule is found that matches: following its definition (see Section 9.5.1.1), it accepts a sub-procedure being migrated to a class. The rule creates a method declaration node as a child of the class declaration node (the target context) and gives the name of the sub-procedure to the method. In this case, the return type is set to void as VBA sub-procedures do not return any value. The method is also defined as static to allow invoking it without an object. Then the rule delegates the migration of all children to the engine.

Step 2 Here, "showName" has no parameter, so the only ASG child is the single statement in the sub-procedure's body. The engine now looks for a *Productive Rule* accepting the statement as source entity and the newly created method declaration node as target context. The rule matching these source and target entities is found at the top level of the model. It is the generic AnyCopy rule (Defined in Section 9.4.2.1), that copies the source sub-tree into the destination context. So it will copy the "expression-statement" node (a grammatical node, see Section 9.3.1) of the sub-procedure. And again, delegate handling of its ASG child (a "function call" reference node).

Step 3 This also falls back to the AnyCopy rule. But this time, it must be noted that the copied node is not valid in a Java model (no function calls in Java, only method calls). This problem will be corrected later by a map directive. The reference remains a function call and points to no declaration node. The engine traces the source entity with the rule and the target entity. (see Section 9.4.2.1).

Step 4 The next step is to handle the ASG children of the function call. In this case, it is the argument of the function: the binary operation. This time CopyReplaceOperator (defined in Section 9.5.1.1) rule is found that matches. This rule has been configured to detect and replace the "&" binary operator (string concatenation) with a "+" operator.

Step 5 and 6 Two new iterations of this process will apply the AnyCopy rule on the operator's arguments. As the first argument, this gives a literal string node (valid in Java) and, as the second argument, a reference to a variable named "name". Note that at this stage, the reference node points to nothing (no declaration node) for the moment. The engine traces the variable reference from the source model with the rule and the target variable reference. (see Section 9.4.2.1).

Step 7 The engine is now done with creating nodes in the target model, the entire AST of the "showName" sub-procedure declaration has been recreated (in a modified form) in the Java model, and there are two references (variable access and function call) pointing to nothing.

The engine then looks at the references that were created. For the variable reference, it searches whether an adaptive rule can make this variable reference fit the target model (it should be pointing to an existing variable). Supposing there is none (the variable still needs to be migrated), the engine creates a stub declaration to which the variable reference can point, which points to the variable reference in the source model. Similarly, for the function call, the engine looks for an existing adaptive rule able to adapt the call to MsgBox to fit the target model (it should be pointing to an existing method) and does not find one. A function declaration stub is created, pointing to the source model's function invocation (information traced in 3 and 6. see Section 9.4.2.1). The state of the models at this stage is pictured in Figure 9.4.

Step 8 Finally, in the context associated to"MyDestination" target class, a mapping is registered from the legacy "showName" function to the migrated "showName" method. At this stage, the produce directive is finished; it gives a target model that is not coherent as it contains nodes that are not valid in Java. Invalid nodes are allowed temporarily if one does not try to export the model to the source code.

Please, note that two features of our meta-model are leveraged in this explanation. (i) Four out of six rule applications use the AnyCopy rule. This is due to our application meta-model rationale based on similarities and differences. (ii) The application of the directive finished in an expected yet inconsistent state. This is possible thanks to the model's ability to represent *inconsistent* intermediate states.

Such is the interest in using a heterogenous unified meta-model, able to copy as default behaviour and represent inconsistent intermediate states.

Both features were explained in Section 9.3.1. Such is the interest in using a heterogenous unified meta-model.

9.5.3 Example of the application of a map directive

Having converted the VBA "showName" sub-procedure to a Java method, the user is notified that this method is incomplete: it requires the resolution of two artefacts in the target model: (i) the invoked method "MsgBox" (ii) the referred variable "name".

For helping to resolve these artefacts, our approach includes an action to manually express that these missing artefacts have an equivalent in the target model: the map directive.

To do this, the user applies a map directive (see Section 9.3.2) that is going to register a mapping between the source declaration of "MsgBox" (a *declaration* that contains no definition as it is a VBA library routine), and the target declaration of Java method "log", within the context of "MyDestination" class. Each time a mapping is registered, either by the map directive or as a result of migrating a

declaration (step 8 Section 9.5.2), the engine starts what we call an *adaptive phase*: the systematic application of adaptive rules.

We break down the map example into five steps, including registering the mapping and the *adaptive phase*; the first three steps are illustrated in Figure 9.5. The application of adaptive rules is made with a kind of "double lookup".

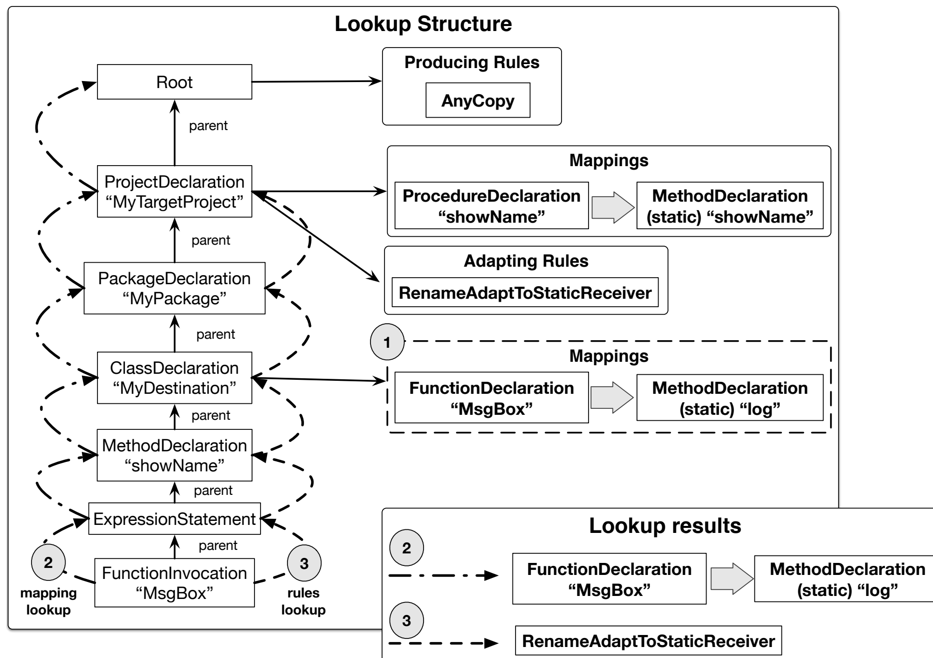


Figure 9.5: Map directive application: Mapping registering, mappings lookup, adaptive rule lookup. The mapping (ProcedureDeclaration "showName" => MethodDeclaration "showName") was automatically registered during the Produce directive The numbers in the visualisation refer to the step related. Step 1 creates a mapping. Step 2 looks up all mappings affecting the function invocation "MsgBox". Step 3 looks up an adaptive.

Step 1 The engine registers the map between the source and target declarations in the list of mappings of the top context of the Java target. Meaning that this mapping is available for all artefacts in this target. (MsgBox => MyDestination.log)

Step 2 First step of the adaptive phase. The engine finds all unresolved uses of the source entity in the target model: all the reference objects pointing to stubs related to the source declaration. The first lookup aggregates all the possible mappings affecting the reference. As shown by Figure 9.5, this lookup starts from the context of the reference. This first lookup yields an ordered list with all possible mappings affecting the analysed reference. The most concrete

mapping is tested first. In our example, the only mapping responding to our case is the one just installed.

Step 3 For each mapping obtained in Step 2, we start a lookup from the reference, looking for a rule which accepts the reference and the mapping.

The second lookup tests all possible adaptive rules with each reference and each mapping in the obtained order. As shown by Figure 9.5, this lookup starts from the context of each reference. This process yields the first rule that tested positive for its application. If no rule is found, the process finishes. In our example, the rule testing positive is RenameAdaptToStaticReceiver.

Step 4 The process finishes if no rule is found. If a rule is found, the engine applies it with the reference and the mapping that was accepted. In our example, the engine applies the RenameAdaptToStaticReceiver parametrised with the function call and the mapping that represents equivalence between "MsgBox" and "MyDestination.log". As an outcome, the rule replaces the function call with a method invocation, using the same arguments as the original call, and setting `MyDestination.log` method as referee.

Step 5 Last step of the adaptive phase. The engine removes all the useless stubs.

In this way, the mechanism can be applied retroactively (by searching for past Stub nodes) and to future migrations (because the produce directive registers mappings for all migrated declarations).

9.5.4 Summary

As told in Section 9.4 the engine must respond to decisions of a software migration depending on the circumstances: technology, developers and users.

- To allow choosing **what** to migrate, we support partial migrations. A produce directive produces the migration of a selected source entity within a selected target context. The user chooses both the source entity and target context. We also support linked stubs (as created in Step 7 of Section 9.5.2) to allow knowing **what** is left to migrate to make work what we have migrated so far.
- To allow choosing **how** to migrate, we use context-aware rules and mappings. According to a specific target, available rules and mappings directly impact our outcome, which is the reason for the lookup in Step 1 Section 9.5.2.
- To allow choosing **when** to migrate and to ensure the **decision consistency**, we support adaptive rules based on mappings and linked stubs, which adapt previously migrated code to fit new decisions, as done in the adaptive phase started in Step 2 and 3 of Section 9.5.3.

9.6 Discussion

9.6.1 Rule morphology

Traditional model transformation approaches aim to be fully automatic ("push button" approach), and reproducible [Siikarla 2008]. For this, the engine applies all the rules to every entity in the source model that it can handle. As such, transformation rules are typically composed of at least two parts (*e.g.* [Czarnecki 2006]): (i) Left-Hand Side (LHS), a selection part specifying which source model element must be matched by the rule; and (ii) Right Hand Side (RHS), a transformation part specifying what target model elements must be generated and how they should be initialised. The LHS part can be tricky to define in these approaches when one needs to precisely pinpoint a subset of entities, or a unique entity, for specific rules.

Our rules share this characteristic: They have an LHS part (what we call condition) that filters the kind of entities they can apply to. But because our rules are started by a user directive on a source entity and have a context of application, the filtering part is usually straightforward, for example, only checking for a specific entity type.

9.6.2 Rule reentrancy and scoping

Reentrancy. A rule can always delegate its application over an ASG node's children. Like this, the rule that can create a Java method from an Access function will rely on other rules to create the parameters of the method or its body.

Scoping. Migration rules work from a source model to a target model. In our approach, a rule's *source model* is often, but not always, the legacy application model. Symmetrically, the *target model* will usually be one of the migrated application models (it is always the case in this chapter). Still, we envision cases where the legacy application model could be the target of a transformation rule, for example, in a partial migration where parts of the legacy and migrated applications run together (see, for example, [Verhaeghe 2022]). In such a case, the legacy application may need to be modified to reference some declaration node in the migrated application.

Rules are defined and applied within a *context*. The context can be the entire application, a specific package, a specific class,... Concretely, the context can be any declaration node in the model of an application [1]. When the context of the rule is the top-level node of an application model, this allows one to handle the specific programming language of this application. For example, one could have

[1] In the current implementation, we store the contexts in a "twin tree" of the target AST (i.e. with the same branches). For simplicity, this paper considers that contexts are stored in the AST.

an "RPC"[2] package with special migration rules to add Spring Boot annotations automatically.

9.6.3 Modelling

In Chapter 6, we discussed the modelling of migrating applications. We introduced the difference between declarations and grammatical entities. The boundary between *declaration* and *grammatical* is not clear cut. It depends on whether we want to be able to handle references on an entity. For example, in Java, primitive types are part of the language's grammar. Still, they are modelled as declaration nodes because entities like variables have reference to them in their declarations and also because we might need to migrate them to other types (*e.g.* Java boolean to C int). Conversely, the "+" operator is currently modelled as a grammatical node. Still, it could be modelled as a function (a declaration node), for example, when this operator represents the string concatenation in Java. In this case, the "+" in the Java expression " `"the answer is:"+42` " would be modelled as a reference to a particular java function *stringConcatenation*, with two arguments, so that one would be able to add migration rules for it.

9.6.4 Migrating approaches

The engine allows an expert to develop rules for applying and defining the operational migration strategy. In our case, we decided to translate code as a baseline and add higher-level rules for dealing with the visual component and its data bindings because the goal of our migration is to abandon the Microsoft Access platform completely.

However, rules for generating black-box or grey-box approaches are doable as well. A simple and naive example of producing a wrapper is the rule applied when applying the produce directive with a controller method from the back end into a service class in the front end.

9.7 Conclusion

Complete system migration is an overwhelming, complex and intimidating mission to achieve.

Our particular case of migration is a case with multiple layers of software migration, as we described in Section 9.2.1. Translate language implies using different grammatical rules for saying the same and using entirely different constructions to think of problems when the paradigm behind them is distant.

[2]Remote Procedure Call: When a method in a program calls a method defined in a different program, usually on another computer

Different languages grow surrounded by different communities addressing problems in different and often incompatible ways, favouring the apparition of different libraries and frameworks, from the definition of their concerns to the definition of their API. Different deploying environments and architectural designs highlight different features and requirements impacting the inbound design in radically different ways, such as the differences between a monolithic application and a microservices one. All these gaps impact at many levels; not only must the code change ultimately its nature: the grammar, the APIs, the concepts and the concerns but the developers as well. We claim that for large mission-critical projects with such a gap, fully automatic migration is almost as undesirable and unreliable as software rewriting since an automatically migrated mission-critical project would have no immediate maintainers nor developers at most of the aspects of the target projects: source code, library, architecture, building, shipping, and deploying. Even from the point of view of the database, which we do not migrate, the code interaction would change, meaning that the knowledge of the database administrators may fail with the new versions of the code.

For tackling down, we propose an iterative, interactive migration process (Section 9.2.2)that is based on three central ideas: developer control, immediate feedback, and independent life cycle of source and target systems. Our proposal has as its major drawback the requirement of good migration planning. Architectural and design decisions have to be taken as in forward engineering. To promote experimentation (which is highly required for taking decisions), we reinforce the importance of not punishing bad decisions by being able to undo all modifications from both model and source-code points of view.

Our approach proposes to scope the kind of migrating approach within the design of each rule. In our migrating approach, language migration is compulsory as the intention is to abandon Microsoft Access completely. Therefore all our rules are based on white-box approaches.

However, this lack of black-box or grey-box approach rules does not mean that the approach is incompatible with it. A simple and naive example of producing a wrapper is the rule applied when applying the produce directive with a controller method from the back end into a service class in the front end.

In Chapter 10, we present three different validations of the approach, targeting three different technologies.

CHAPTER 10

Case of studies: Migrating language, paradigm, libraries, GUI and architecture from Microsoft Access

Contents

10.1 **Semi-automatic form migration to Typescript Angular front-end and Java Springboot back-end** 142

10.2 **Migrating tables and queries to a Java back end and a Typescript front end** 161

10.3 **Automatic library and paradigm use migration from Microsoft Access to Pharo and Java** 164

10.4 **Conclusion** 177

This chapter presents three validations of the migration approach in Chapter 9. Along with the validations, we analyse each migration's specific constraints and requirements. With this analysis, we also introduce the rationale behind the rules. Some rules are simple language replacements (such as CopyAsStaticMethod); others require more complex strategies, including using knowledge models to represent GUI and data bindings.

Section 10.1 presents the validation of the iterative, interactive approach to migrating an MS Access form to the front and back ends. Section 10.2 presents the validation of the iterative, interactive approach to migrating MS Access tables and queries to the front and back end. Section 10.3 presents the validation of the iterative, interactive approach to migrating MS Access library and procedural-paradigm uses to Pharo and Java.

10.1 Semi-automatic form migration to Typescript Angular front-end and Java Springboot back-end

In this section, we migrate a Microsoft Access Form to Typescript and Angular front-end and a Java and Springboot back-end. Section 10.1.1 motivates this case of study. Section 10.1.2 presents the form we aim to migrate, presenting its code and look. Section 10.1.4 lists the rules required for this experiment. Section 10.1.6 exposes and analyses de results and what was needed to make the targets compile and execute. Section 10.1.7 discuss some details of the results.

10.1.1 Motivation

We present a real case study of the approach presented Chapter 9. The motivation is to have a simple migration including all the migrating aspects of our industrial case of migration: Splitting the application into the front end and the back end. Produce a back end that provides the data required by the front end. Produce a front end that consumes data the back end provides and shows an equivalent UI.

10.1.2 The login form

We fully migrated the login form MS Access Northwind Traders (depicted by Figure 10.1) to Java+Springboot and Typescript+Angular.

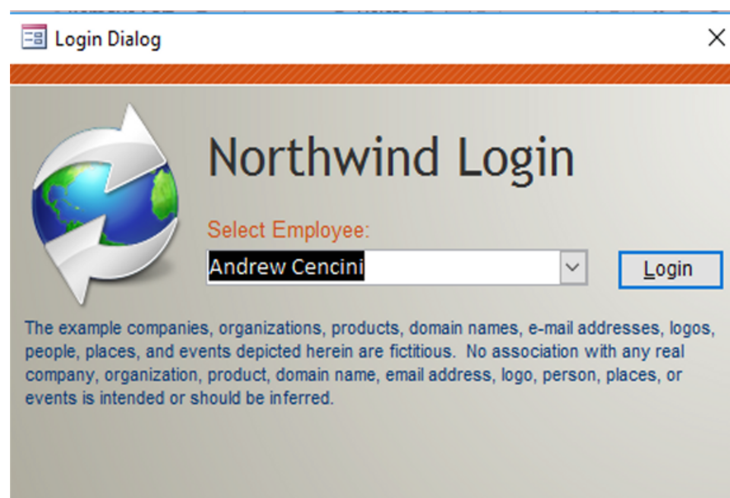


Figure 10.1: Form to migrate

Listing 10.1 features the code that manages this form. It counts with two sub-procedures for managing specific events of the form.

```
public Sub cboCurrentEmployee_AfterUpdate()
  On Error GoTo cboCurrentEmployee_AfterUpdate_Err
  Call TempVars.Add("CurrentUserID", Screen.ActiveControl.Value())
  On Error Resume Next
  Call DoCmd.Requery("")
  Call DoCmd.RunCommand(acCmdRefresh)
  cboCurrentEmployee_AfterUpdate_Exit:
  Exit Sub
  cboCurrentEmployee_AfterUpdate_Err:
  Call MsgBox(Error())
  Resume cboCurrentEmployee_AfterUpdate_Exit

End Sub

public Sub cmdLogin_Click()
  On Error GoTo cmdLogin_Click_Err
  If Not IsNull(cboCurrentEmployee.Value) Then
    Call TempVars.Add("CurrentUserID", cboCurrentEmployee.Value)
    Call DoCmd.Close("")
    Call DoCmd.OpenForm("Home", acNormal, "", "", acNormal)
    Exit Sub

  End If

  Beep
  Call MsgBox("You must first select an employee.", vbOKOnly, "")
  cmdLogin_Click_Exit:
  Exit Sub
  cmdLogin_Click_Err:
  Call MsgBox(Error())
  Resume cmdLogin_Click_Exit

End Sub
```

Listing 10.1: Login northwind

The login form source code (Listing 10.1) has two sub-procedures handling events.

cboCurrentEmployee_AfterUpdate. This sub-procedure is activated after the drop-down list selection changes. Line 2 says that whenever there is an error, we should execute the code in line 10 and then in line 8. Line 3 says to store as a temporary global variable the value of the active control (which is the one that just changed: the combo box). Line 4 overrides the error management policy. Whenever there is an error, we should execute the following statement. Lines 5 and 6 force the re-query of all the form data bindings and then force the graphical updating of the form. Line 8 finishes the execution of the sub.

cmdLogin_Click. This sub-procedure is activated after clicking the login button.

The lines, including novelties, are: Line 17 uses an if with an IsNull function call in the condition expression. Lines 19 and 20 are navigation commands: close the form and open a "Home" form. Lines 25 and 26 manage cases where a non-user has been selected using Beep (a statement producing a sound) and MsgBox (function invocation informing the user of the lack of selection). Finally, Line 30 informs the user about an error during the execution.

10.1.3 Understanding the example in terms of our targets

For our example, we have two main targets of migration: The Java SpringBoot back-end and the TypeScript Angular front-end.

To understand the requirements of the target systems, we should first understand what code we should migrate from the architectural point of view of the targets.

Front-end. For the running experiment, we decide that from the visual point of view, the target should be identical to the source as shown by Figure 10.1.

Architectural decisions: From the behavioural point of view, we decided that the management of the temporary global variable stored in *"TempVars"* (lines 3 and 18) should be managed by the backend. From the event handling point of view, the destination drop-down list and button should be related to equivalent event handlers. Finally, from the data usage point of view, the cboCurrentEmployee drop-down list must be filled up with data from the backend. We must analyse their code to decide how to translate it on the sub-procedures.

Language and library migration. From the language point of view, the On Error Go To used in both sub-procedures, we propose to replace it with a try-catch-finally.

For storing session information, such as the setting of a global variable, we consider it a back-end concern, meaning that both usages should be rephrased to contact the backend instead of using a local class.

Line 4 is a challenging line to solve. We have no solution for it other than translating it as a comment since the required code to emulate the same semantics, even when possible, would be unreadable.

For the refreshing of the content proposed by lines 5 and 6, we consider that most of the time, a programmatic refresh invocation will be undesirable because it stresses the backend and network communication. Therefore we propose to avoid its translation or maybe translate it as a comment since it could give further information to the developers.

For the if statement (line 17), we propose copying. We could propose a rephrase for using the function IsNull if we find an equivalent behaviour in the target technologies. Instead, we propose to migrate it as an equals-to-null binary operation.

Lines 19 and 20 require more understanding of the next steps in the migration.

We propose to comment on them in this example, but we should replace them with some navigation function or redirection.

Line 25 with the beep statement, we propose completely dismissing it since using a beep sound is unacceptable on our front end. Lines 26 and 30 are both on the usage of MsgBox. If we want to keep the popping-up behaviour for showing the error, we can map MsgBox with the alert function.

GUI and infrastructure migration. In this case of migration, our target form should look like the original one developed in MS Access.

To accomplish this migration, we implement a rule using a *Casino* model (see Chapter 4). Casino was proposed by Verhaeghe *et al.* [Verhaeghe 2021b]. It consists of a meta-model representing the visual reality of a GUI widget representing different graphical elements *e.g.* screen, windows, layouts and controls (text boxes, drop-down lists, etc.). This model helps us to organise the visual knowledge into a coherent model that we can use to produce the proper Typescript structures and HTML visual compositions.

Microsoft Access GUI controls can be configured to load data from given SQL queries or table names. We extended the Casino meta-model to add data dependencies. For example, we annotate the element representing the cboCurrentEmployee drop-down list with a property "data source" holding a SQL query description.

We extended the basic model of Casino also to link graphical elements to the existing ASG code as specific event handling, which can be translated into the target Typescript-Angular component. For example, we annotate the element representing the cboCurrentEmployee drop-down list with a property "OnChange" holding the source model definition of the `cboCurrentEmployee_AfterUpdate` sub-procedure.

For translating the behavioural code, which is in charge of responding to the different events to handle the target widget, we define some language translation rules.

Back-end. Taking into account the decisions of the front end and analysing our running example, we find two primary responsibilities for the backend:

(i) The login combo box must be filled with data. Meaning that somehow we need a backend able to respond with the information required to fill up such a form.

(ii) When the user is chosen, we must set a user id in a temporal variable. For the sake of the example, we decided to replace the idea of a temporal variable with a session variable on the server side. None of these responsibilities can be translated from existing written code. Along with that, the way to organise the required code is highly constrained by the chosen technology and its best practices.

To replace the infrastructure provided by Microsoft Access, we consider our target technology.

Our back-end project works with Java as a language. Still, it also delegates to SpringBoot to define what "an application" is and how to process HTTP requests, including the type marshalling for exchanging information with the front end our

eventually other backends. SpringBoot ships within other frameworks, such as Spring (for dependency injection) and Hibernate (ORM).

We must use these frameworks and libraries to migrate the library and infrastructure to the back end.

This means we must use Hibernate and Spring Boot to fill the login combo box in the front end. This has further implications: (i) we need Data Access Objects (DAO) to access the library that Microsoft Access transparently with its infrastructure; (ii) we need to ensure that the query used to fill the drop-down list is accessible in some DAO. (iii) we need object-oriented Data Transfer Objects (DTO) to hold values from a table and read them in the back end, and transfer them to the front end; (iv) we need a Spring Boot controller and service that exposes the data to the front end.

10.1.4 Rules and mappings

We follow to explain the dynamics of the definition of rules and mappings based on our experience.

Let us consider that we start only with the AnyCopy rule, applicable to any construction to be regenerated in any chosen target.

If we only leave this rule, migrating a Form to any destination will give us the same: a Microsoft Access Form. But as it is in Java or Typescript, it yields a wrong result. We must start defining rules that allow us to produce constructions that make sense in each destination.

10.1.4.1 Language translation productive rules.

We started with the productive rules required to translate code from Access to Java and Typescript.

Language translation rules are based on mapping one language's syntactic structure with an equivalent of the destination. We present the rules and detail some of them.

- Java:

    1. CopyAsPersistentClass: Any Table or Query migrated into a package is defined as Java-Class persistent entity.

- Typescript:

    1. CopyAsDTO: Any Table or Query migrated into a package is defined as a Class to use as DTO.

- Common

1. CopyAsClass: Any Module migrated into a package is defined as a Class.
2. CopyAsStaticAttribute: Any variable belonging to a module migrated into any class is defined as a static attribute.
3. CopyAsStaticMethod:
   - Any sub-procedure belonging to a module migrated into a class is defined as a static method with void as returning type.
   - Any function of a module migrated into a class is defined as a static method.
4. SimulateSetToReturn: Any body of a function migrated as the body of a method must add a temporary returning variable and a returning clause at any cutting point of this function.
5. CopyReplaceOperator:
   - Any usage of = as equality must be defined as ==.
   - Any usage of <> must be defined as !=.
   - Any usage of Not must be defined as !.
6. CopyAsTryCatch: Any usage of On Error Go To that responds to a translatable pattern may be defined as Try/Catch.
7. CopyAsComment:
   - Any usage of On Error Go To that cannot be transformed into Try/-Catch must be commented.
   - Any usage of "label:" must be commented.
   - Any usage of DoCmd.Requery, DoCmd.RunCommand, DoCmd.Close or DoCmd.OpenForm must be commented.

CopyAsStaticMethod and CopyReplaceOperator have been explained in Chapter 9.

CopyAsStaticAttribute is similar to CopyAsStaticMethod. CopyAsComment migrates all the nodes into a comment node.

CopyAsPersistentClass. Any Table or Query migrated into a package is defined as Java persistent entity. A persistent entity is an object-oriented representation of what a row contains in a table. These kinds of objects are used when accessing the table content. Each row is going to be fetched as an instance of this class.

Context: Java migrated application

Condition:

1. The source entity is a table or a query.

Operation:

1. Defines a class using the same name as the source entity.
2. Annotates the freshly created class annotated with a javax.persistence.Table, configured with the name of the source entity.
3. Annotates the freshly created class annotated with a javax.persistence.Entity annotation.
4. Each column in the source entity is migrated as an attribute.
5. Each attribute created from a column migration is annotated with a javax.persistence.Column configured with the name of the column.
6. If the column is marked as PrimaryKey, the attribute created for this column is annotated with a javax.persistence.Id annotation.
7. Annotates all the added children as Java persistent attributes.
8. Generates setters and getters.

CopyAsDTO. Any Table or Query migrated into a package is defined as Java persistent entity.

Context: Typescript migrated application

Condition:

1. The source entity is a table or a query.

Operation:

1. Defines a class using the same name as the source entity.
2. Each column in the source entity is migrated as an attribute.
3. Generates setters and getters.

CopyAsClass. Any Module migrated into a package is defined as a Class.

Context: Java or Typescript migrated application.

Condition:

1. The source entity is a module.

Operation:

1. Defines a class with the same name as the source entity.
2. Migrates all the children of the source within the freshly defined class

CopyAsTryCatch. All usage of On Error Go To that responds to a translatable pattern may be defined as Try/Catch.

We can only translate those ON ERROR GO TO Label as try-catch, which statements can be split into independent code blocks.

Context: Java or Typescript migrated application.

Condition:

1. The source entity is a Block **AND**
2. Target is a Method **AND**
3. Contains On Error Go To Label **AND**
4. No Go To Used before Label **AND**
5. Just before Label, there is a terminating statement (Exit or Return)

Operation:

1. Define a block inside the target Method
2. Migrate all the statements before On Error Go To and Label into the method block
3. Define a Try/Catch statement into the method block
4. Migrate the children between On Error Go To and Label into the try block
5. Migrate the children between Label and the end into the catch block

SimulateSetToReturn. Any body of a function migrated as the body of a method must add a temporary returning variable and a returning clause at any cutting point of this function.

Context: Java or Typescript migrated application.

Condition:

1. The source entity is a block **AND**
2. The source parent is a function **AND**
3. The target is a method **AND**

Operation:

1. Define a block in the target method.
2. Add at the beginning of the migrated block a temporary variable typed as the returning type of the method.
3. Migrate the children of the block into the freshly created block.
4. Add a return at the end of the block, returning a reference to the just added variable.

Language translation productive rules summary. After writing these productive rules, any module would be translated as a class. Classes are translated as classes. We can also translate sub-procedures and functions as methods considering their different details.

There are still two main problems with these rules: (i) None of these rules solve the problem of accessing or invoking things at our targets. Function and sub-procedure calls stay the same, even though java requires method invocations. (ii) These rules do not comply with the expectations: no back-end DAO nor front-end component can be created with these rules.

Adaptive rules. We could choose to transform the usage of artefacts as a language translation. If a function is translated as a static method, we translate all the usages as static or self-method invocations. The problem with that approach is that it assumes that the functions or procedures will be translated. (but it may not be the case for functions that make no sense on the destination or for third-party libraries). It also assumes that the functions and sub-procedures will be translated before translating any of its usages (something that would seriously affect the user's interactivity).

To deal with this, we are going to leverage the adaptive rules. Like this, we can allow translating the usage of an artefact that does not exist by keeping their shape (function call, variable access, etc.) in exchange for adapting them when the artefact is migrated, or a semantical equivalent is provided by using the map directive.

We will rely on two adaptive rules: SimpleRename and RenameAdaptToStaticReceiver.

SimpleRename – Replace any reference name with the name of the artefact. This rule is in charge of adapting all the types based on mappings. The rule SimpleRename is activated when the object reference does not need any modification to refer to a target proposed by a mapping. Still, the name provided by the reference differs from the one on the mapped target declaration. In that case, the object reference name is modified to respond to the same name given to the target declaration. This rule is mainly applied when migrating types. Let's consider a map between the types dbText from MS Access and String from Java. The Rename rule would modify any variable declaration using dbText to use String.

Context: Java / Typescript

Condition:

1. The target entity is a reference **AND**
2. The target entity can refer to the mapped target declaration **AND**
3. The target entity name differs from the mapped target declaration name.

Operation:

1. Set the name/selector of the mapped target declaration to the target reference.
2. Set the target entity to refer to the mapped target declaration.

The second rule, RenameAdaptToStaticReceiver, is explained and detailed in Chapter 9.

10.1.4.2 Knowledge-based productive rules.

To define the behaviour of a form in Microsoft Access, we use source code, but we also use meta-data. With meta-data, we can define the visual aspect and access to data of a form.

Forms visual definition A Form does have as well lay-outing strategies and visual characteristics. Finally, a form has the capacity to process the different events that may occur during the application usage, such as button clicks, combo-box selections, etc.

All this information is entirely invisible from the code point of view and is resolved by Access based on conventions or declarative devices: *e.g.* there is no setting of the event handlers as such, there is no code connecting to the database, there is no CSS defining a layout.

Forms data access definition A Form default instance can be configured to draw data from a specific Table, Query or SQL Query at the level of the form or the level of any composing graphical elements, such as a table or a combo box, establishing some data dependency. After configuring a Form with a table or a query, the form will automatically access this information. This information is accessed by field name, as when working directly with recordsets. This means the code using this data is developed depending on a row structure.

Knowledge Models. To deal with this infrastructure migration, analyse and combine information related to the form into a pivot or knowledge model. This model is used later as a specification to produce target ASG modifications.

We use two knowledge models and their eventual combination to implement rules able to migrate GUI and data-access infrastructure. Most techniques applied here are inherited from the Model transformation domain. Before detailing the rules specifications, we introduce these knowledge models.

Casino model was proposed by Verhaeghe *et al.* [Verhaeghe 2021b] (see Section 10.1.3).

We produce a Casino model by analysing the form's ASG. This is why it is also called a pivot model. In our use case, the model ships the methods of the

source Form, which handles specific interaction events, to be able to generate the destination code by reusing the previously defined language translation rules.

DataBinding model is proposed by us. It is a simple model based on extracting the different parts of an SQL query obtained from the Form properties. We extract the table name and columns by analysing the query. With this, we produce an object which holds the table name, the columns, and the query string. We annotate Casino objects with these data-binding objects to represent the binding between GUI controls and data. In our example, the combo box cboCurrentEmployee is configured with a data source property as shown in Listing 10.2.

```
SELECT *
FROM Employees_Extended
ORDER BY Employee_Name;
```

Listing 10.2: Data Source cboCurrentEmployee

As we link the different GUI controls with data, we can produce target GUI components that consume remote information to load the different controls of the widget.

- Java
    1. CopyAsDAO: All Table or Query migrated into a DAO package is defined as a DAO Class for managing this specific table/query entity
    2. CopyAsDAOMethod: All Form migrated into a DAO package is defined as methods in all the DAOs that should be used for acquiring the data the Form uses.
    3. CopyAsService: All Form migrated into a Service package is defined as a Spring Boot Service class that delegates all data access to all the related DAOs
    4. CopyAsController: All Form migrated into a Controller package is defined as a Spring Boot Controller class that defines request endpoints and resolves them by delegating to the appropriated Service.
- Typescript
    1. CopyAsComponent: All Form migrated into an Angular Module is defined as an Angular Component and an Angular Service.

Knowledge-based productive rules specifications.

CopyAsDAO. Tables and queries are database-related concepts. In our back-end target, we represent access to databases by using DAOs. A DAO is an object

that creates, reads, updates and deletes entities from a table. When manipulating data from the database, the DAO requires another class representing the content of a table. We distinguish these classes because they are annotated with a javax.persistent.Table annotation and this annotation holds the name of the same table or query we are migrating.

Context: Java / DAO package.

Condition:

1. The source entity is Table or Query.
2. It exists a class annotated with a javax.persistent.Table annotation **AND**
3. The annotation holds the name of the same table or query we are migrating.

Operation:

1. Generate a DAO class, which accesses elements of the type produced by the rule CopyAsPersistentClass. (Remember CopyAsPersistentClass creates objects that represent the contents of a table).
2. Add standard methods: list, save, update.

CopyAsDAOMethod. Forms and form controls can be bound to queries. This binding is used to fill up controls and forms with data from the database. When migrating a form into a DAO package, we analyse all the bindings within the form. We find the main table or query used to access information for each binding. We find the DAO accessing the data of the used table or query. We modify the related DAO by adding a method that fetches the same information as the query used in the original form.

This rule works only with queries using a single source (table or access query).

Context: Java / DAO package.

Condition:

1. The source entity is a form **AND**
2. The form has data bindings **AND**
3. Each binding points to a single table or query **AND**
4. All tables and queries can be accessed by using one DAO.

Operation:

1. For each binding extracts the table's name or query.

2. For each table or query name, it searches for an existing DAO providing access to it.
3. For each binding, a method encodes the query required by the form in the DAO found in the previous step.

CopyAsService. Any data accessed by a form must be accessible through a service. We aim to have one service by migrated form. Each service may access many DAOs.

Context: Java / Services package.

Condition:

1. Source is a Form **AND**
2. The form has data bindings **AND**
3. Each binding points to a single table or query.

Operation:

1. For each binding extracts the table's name or query.
2. For each table or query name, it searches for an existing DAO providing access to it.
3. For each binding, create a method annotated as service, delegating to the selected DAO.

CopyAsController. Any data accessed by a form must be exposed as a service endpoint. For doing through a controller. We aim to have one controller by migrated form. Each controller uses a related service.

Context: Java / Controller package.

Condition:

1. The source is a form **AND**
2. The form has data bindings **AND**
3. Each binding points to a single table or query.

Operation:

1. For each binding extracts the table's name or query.
2. For each table or query name, it searches for an existing service providing access to it.
3. For each binding creates a method annotated as controller, delegating the selected service.

CopyAsComponent. When a form is migrated to our front-end, we want to produce a GUI Angular component with the same visuals and behaviour as the original form.

Context: Typescript / Angular module.

Condition:

1. Source is a Form.

Operation:

1. Extracts an extended Casino model out of the form's ASG.
2. Using this model, it generates an Angular component and a related service.

Knowledge-based productive rules summary. While language translation rules act on the ASG structure by copying and replacing language concepts, knowledge-based rules combine information into a descriptive model. This model is used to modify the target ASG responding to the same model.

After writing these productive rules, we can produce the expected target GUI widgets and the required infrastructure to transmit data from the back end to the front end.

There is still a main issue that is not addressed by these rules: There is no communication between the front and back ends.

10.1.4.3 Back-end front-end linking rules

We use rules to establish connections between the back and front ends.

These rules tweak the migration of elements according to specific targets and sources: one is specific to the backend, defining endpoints, and the other is specific to the front end, defining calls to the backend.

AutoAnnotateController. Any method produced into a spring boot controller class due to an individual migration must be annotated as a controller endpoint.

Context: Java Springboot controller class.

Condition:

1. Source is a behavioural entity (function, subprocedure or method).
2. Target is annotated as Springboot controller class.

Operation:

1. Defines a method with the same selector as the source entity in the target entity. If the source entity is a subprocedure, the method is set to use void as returning type reference.
2. Annotates the produced method as a controller endpoint.
3. Migrate all the children of the source entity using the method as the target context.

The outcome of this rule is that whenever the user migrates a function into a Springboot controller, this function will be translated as a method annotated as a Springboot endpoint.

CopyAsCallbackToBackend. Any method produced into an Angular service class due to an individual migration of a spring boot controller endpoint must produce a method with the same signature, implemented as calling back to the backend.

Context: Typescript Angular service class.

Condition:

1. Source is a behavioural entity (function, subprocedure or method).
2. Source is annotated as controller.

Operation:

1. Defines a method with the same selector as the source entity in the target entity. If the source entity is a subprocedure, the method is set to use void as returning type reference.
2. Migrate all the children *except the body* of the source entity using the method as the target context.
3. Writes the body of the method by adding a single statement returning the value evaluation of a remote call using the controller annotation information.

The outcome of this rule is that whenever the user migrates a controller endpoint (method) into an Angular service, the engine will produce a method within the Angular service with a migrated signature, and the behaviour is to invoke the given endpoint remotely.

10.1.4.4 Mappings

The mappings for this experiment are presented in Table 10.1.

Table 10.1: Mapping examples table

VBA	Java	Typescript
Access.dbText	java.lang.String	string
Access.dbMemo	java.lang.String	string
Access.dbDate	java.time.LocalDate	Date
Access.dbAttachment	java.sql.Blob	any
Access.dbInt	int	number
Access.dbDouble	double	number

10.1.5 Migrating the login example

The Login form depends on the Employees_Extended query. So we decided to start by migrating it to both the back and front end.

Our rules are based on a naming convention to be able to ease the generation of code.

Migrating Employees_Extended. *Migrating into back-end at the model package* activates the rule CopyAsPersistentClass. It generates the class named Employees_Extended_Model annotated as a java persistence entity. *Migrating into back-end at the DAO package* activates the rule CopyAsDAO. It generates a DAO class named Employees_Extended_DAO with basic methods such as save, update, and getAll. *Migrating into front-end* regardless of package or module, it activates the rule CopyAsDTO. It generates a class named Employees_Extended_Model with the columns as attributes.

Login form: Back-end. The example becomes more interesting with the migration of the form Login. *Migrating into back-end at the DAO package* activates the rule CopyAsDAOMethod. It analyses the data usage and learns that the cboEmployee is configured with a SQL query related to the Query Employees_Extended. This adds the method listed in Listing 10.3 in the Employees_Extended_DAO class *Migrating into back-end at the Service package* activates the rule CopyAsService. It generates a SpringBoot Service class Login_Dialog_Service, which implements a method which delegates to the recently generated DAO extension. *Migrating into back-end at the Controller package* activates the rule CopyAsController. It generates a SpringBoot Controller class Login_Dialog_Controller, which implements a method that delegates to the recently generated Service class

```
1 public List<Employees_Extended_Model>
2 getAllEmployees_ExtendedOrderedByEmployee_Name() {
3   return entityManager.createNativeQuery(
4     "SELECT * FROM Employees Extended Order By Employee Name",
5     Employees_Extended_Model.class);
6 }
```

Listing 10.3: Method added into DAO based on Login

Login form: Front-end. *Migrating into front-end at an Angular Application* activates the rule CopyAsComponent. This rule applies two operations: (i) produces an angular Service class Login_Dialog_Service with a method generated based on the analysis of the data usage, which will delegate to the backend with a proper call. (ii) it generates an Angular component. A simplified code version is listed in Listing 10.4. The component receives the service as a parameter, and it configures the component to obtain the data from the backend during the initialisation. It also automatically migrates all the event-handling methods into the newly generated component.

```
1  export class login_dialog implements OnInit {
2    public ngOnInit() :void {
3      this.login_dialog_service.getAllEmployees_ExtendedOrderedByEmployee_Name().subscribe((
         data) => {
4        this.getAllEmployees_ExtendedOrderedByEmployee_Name = data;
5      }
6      );
7    }
8    public cboCurrentEmployee_AfterUpdate() :void{
9      try{
10       TempVars.Add("CurrentUserID", Screen.ActiveControl.Value());
11     }
12     catch (error){
13       alert(error);
14     }
15   }
16   public cmdLogin_Click() :void{
17     try{
18       if (not((cboCurrentEmployee.Value) == Null)){
19         TempVars.Add("CurrentUserID", this.cboCurrentEmployee);
20       }
21       alert("You must first select an employee.", vbOKOnly, "");
22     }
23     catch (error){
24       alert(error);
25     }
26   }
27 }
```

Listing 10.4: TS Angular Component generated based on Login

The result includes the usage of TempVars.Add a method that does not exist on the destination. To solve this problem, what we can do with the current setup is to migrate the library method TempVars.Add into the Login_Form_Controller class. This will produce an empty method properly annotated to be used as a controller, as listed in Listing 10.5. The implementation of this method is the responsibility of the user.

```
1  @RequestMapping(value = "StoreSessionValue/{Name}/{Value}")
2  public void Add(@PathVariable String Name, @PathVariable String Value) {
3    /* Empty */
4  }
5
```

Listing 10.5: Method translated into back-end Controller from Add function

```
1  public Add(Name : String, Value : String): void {
2    this.http.get(`${this.baseUrl}/StoreSessionValue/${Name}/${Value}/`).subscribe((
      data)=> data);
3  }
```

Listing 10.6: Method translated into front-end Service from Add controller in the back-end

```
1  public cmdLogin_Click() : void {
2      if (not((cboCurrentEmployee.Value) == Null)) {
3        this.login_dialog_service.Add("CurrentUserID", this.cboCurrentEmployee);
4      }
5      alert("You must first select an employee.", vbOKOnly, "");
6    }
7    catch (error){
8      alert(error);
9    }
10 }
```

Listing 10.7: cmdLogin_Click rephrased to use service Add

After migrating this Add controller to the Login_Dialog_Service in the front-end, it will generate the call to the back-end listed int Listing 10.6. Finally, if we map the method TempVars.Add to Login_Dialog_Service.Add cmdLogin_Click and cboCurrentEmployee_AfterUpdate will be both replaced to delegate to the service. We note that these same rules can be used to generate wrapping methods, techniques leveraged by black-box and grey-box methods of software migration.

10.1.6 Results

Figure 10.2 depicts the migrated angular component shown in a browser.

Figure 10.3 shows the result of manually executing the request for information to the backend.

Making it work. The outcome of this example is not immediately compiling and working. We could achieve this by adding more rules, but we decided to go through the last mile by hand. The modifications done by hand are the following:

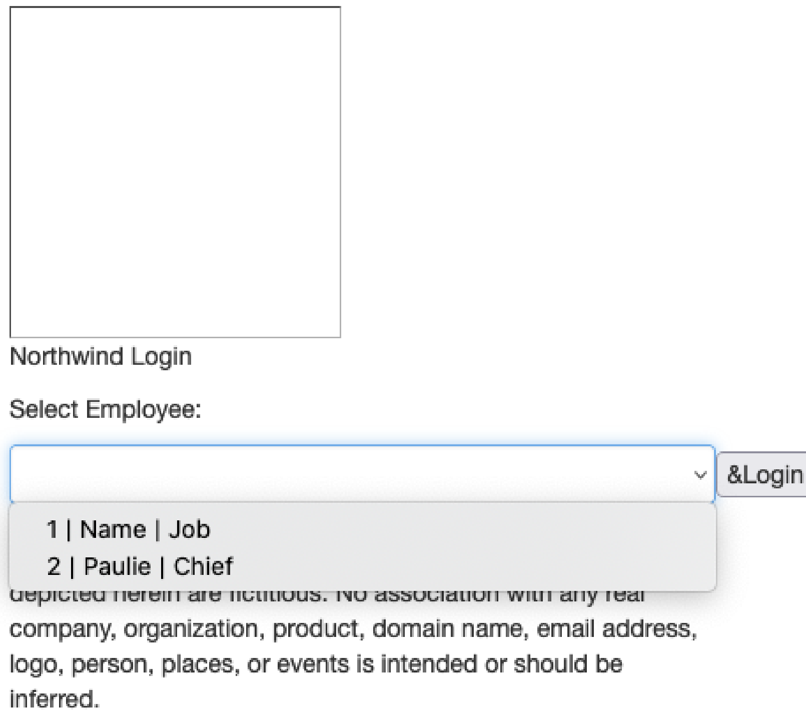


Figure 10.2: Generated Front End

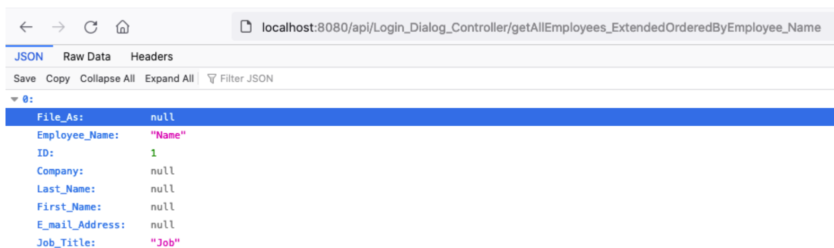


Figure 10.3: Generated Back-End API request

Back end Adding imports (some imports were not calculated), fixing the column names (in access names with special characters are allowed. Our test runs in a Postgres database, which does not allow this).

Front end Adding imports, configuring the angular modules and routing to add the component to the website and changing a call to a method that does not exist by one that exists (to do it automatically would require a new adaptive rule), adding two functions required to fill up the HTML select control with the data coming from the back end: a function that extracts the ID from the DTO, and a function that gives a string to display from the same DTO.

10.1.7 Discussion

We can see that the window follows a similar layout to the original and has the same amount of components. It does not follow the same visual configuration since, for this experiment, we did not make any effort to migrate the GUI design. However, we can access such information from the source model, and the Casino visual model is designed to handle it.

The backend includes a controller handling the login end-point. The implementation we provide now is just an exception announcing that a method must be implemented. In the source application, the click-handling procedure delegates the information to a library function. Our backend implementation handling is a method that responds to the same signature as the source library function but has no other behaviour than a runtime exception prompting the developer to implement its behaviour.

10.2 Migrating tables and queries to a Java back end and a Typescript front end

In this section, we migrate all the tables and queries of the Microsoft Northwind project [1]. Northwind consists of 20 tables and 27 queries. Section 10.2.1 explains our requirements for migrating tables and queries to the front and back ends. Section 10.2.2 lists the rules and mappings required for this case. Section 10.2.4 presents the results for migrating Microsoft Access tables and query entities to Java and Typescript. Section 10.2.5 discuss the results and possibilities to migrate the tables and queries differently.

10.2.1 Migrating data access

As discussed in Section 10.1.4.2, the information obtained from the database responds to the row structure of the table or query accessed.

[1] Microsoft Access Northwind traders, learning example

Tables and Queries in the back end. When migrating to a back-end application, we must produce two different results per each table or query: (i) a class to access the data from the database and (ii) a class to store and transfer this data. Along with this, one of the requirements for the migration is to use high-level database accesses to ease the produced code. We propose to produce Data Access Objects (DAO) and Data Transfer Objects (DTO) for each table or query. One of the critical decisions of our DTO is not to reify relations (as we would do in any other project) but only to reify the rows, to reduce the complexity of the migration of the code which uses this data. In our specific case, we will produce a Java Persistence configuration.

Tables and Queries in the front end. When migrating to a front-end application, we produce a single result: the DTO. The DTO in the front end will respond to the same constraints as in the back-end application, as we also want to reduce the complexity of the migration of the front-end code which uses this data. We also want to ease the communication between the front and the back end.

10.2.2 Rules and Mappings

Rules. The *productive* rules, in this case, differ from the Java back end and Typescript front end.

Back end

CopyAsPersistentClass Defines a class with persistent mappings based on the given Table or Query. It adds public accessors.

CopyAsDAO Defines a DAO which uses the class produced by the previous rule. It defines default methods for CRUD actions.

Front end

CopyAsDTO Defines an exported public class based on the given Table or Query. All the attributes are public for accessing the content directly.

These three rules have been presented in Section 10.1.

The *adaptive* rule is the same for both the front and back end: SimpleRename, presented in Section 10.1.

Mappings. The mappings for this experiment are the same as those presented in Table 10.1.

10.2.3 Migrating engine setup

In this part, we only enumerate the defined rules detailing in which context the rules are installed.

Front-End rules setup.

Productive

1 CopyAsDTO.

Adaptive

1 SimpleRename

Back-End rules setup.

Productive

1 CopyAsDAO.

2 CopyAsPersistentClass.

Adaptive

1 SimpleRename

10.2.4 Results

Table 10.2: Results of migrating tables and queries to Java and Typescript.

Kind	Number	Java DTO	Java DAO	Typescript DTO
Tables	20	0 errors	0 errors	0 errors
Queries	27	27 errors	0 errors	0 errors

Table 10.2 shows the results of the experiment.

On the *Java* part of the experiment, we want to note that the queries generated by DTO were not compiling out of the box because they lack the javax.persistence.@Id, which is required for entities. This is because our rule only checks for the primary key in the direct object, which is only possible in tables. We should specialise in the rule to learn which queries can be mapped as entities and which not. And for those that can be mapped as entities, we must infer the primary keys from which tables and make them into a composite primary key for our model. We solved this problem by manually adding the annotations.

We tested each DAO manually, having successful results, which means that the code parses, compiles and executes as designed.

On the *Typescript* part of the experiment, we successfully produced all the required classes without any parsing or compiling errors.

10.2.5 Discussion.

As we exposed in our migration in Section 10.2.1 we do not foresee mapping java persistence for using a full object model since migrating code is primarily used as records. This is why in the back end, we create one model entity per table or query (to be used as a data transfer object – DTO) and a data access object (DAO) to list all the elements, save, update and delete. For the front end, we only need to define the DTO objects to be able to consume information from the back end.

Nevertheless, as we can access all the information related to the table's relations (foreign key, primary key and indexes), and as we can produce arbitrarily complex classes, we have no reason to think that it would not be possible to write rules producing a related *object* model instead of a *row-like* model.

10.3 Automatic library and paradigm use migration from Microsoft Access to Pharo and Java

In this section, we present the validation of the approach by implementing the rules required to instrument mapping-based automatic Library and Paradigm usage migration. Section 10.3.1 presents the problematic and interleaving of language migration with library and paradigm use migration. Section 10.3.2 presents five generalised adaptive rules. Section 10.3.3 presents the project for validating, explaining how this project is crafted based on the analysis of the metrics presented in Section 8.3. Section 10.3.4 presents the experiment context, explaining steps and configurations to migrate the project to Pharo and Java. Section 10.3.5 presents the results of the validation experiment. Section 10.3.6 presents the threats to our validation.

10.3.1 Software migration context

During language migration, we are required to change the language of a program: to express the source program semantics in a new language. Language translation has been addressed as a compiling problem [Aho 1972, Aho 1986, Appel 2002]. Software migration is not compiling, but many techniques can be leveraged in our favour. This has been done in the past for conducting successful language migrations. [Kontogiannis 1998, Brant 2010, El-Ramly 2006, Martin 2002a, Trudel 2012, Malabarba 1999].

Library migration. Translating is more than changing the language. Changing native types, primitives and libraries are compulsory. Terekhov *et al.* [Terekhov 2000] points out how this problem has been neglected in language migration. Despite all these notices and recognition, to our knowledge, there is no article dealing with this issue systematically.

Even considering languages with similar grammatical constructions, languages often run on different environments with different SDK, libraries and best practices. These differences lead to substantially different ways to define APIs and compose and reuse code. After migrating, we expect to have the same behaviour and to use the target environment's native types and primitives, SDK, and libraries: all of which make this environment a desirable target.

Paradigm use migration. We must consider the migration from a procedural paradigm (VBA) to an Object-Oriented paradigm (Java, TypeScript or Pharo). This is a complex issue that requires defining correct abstractions (classes) from variables and functions scattered in a procedural application (*e.g.* [Duddy 2004, Fox 1997, Grosse-Kunstleve 2012, Zou 2001]). There is no well-accepted solution for this, and we left this issue to the migration tool user. This means the user is responsible for abstracting classes from the legacy application and migrating the relevant variables and functions to their appropriate classes. As we saw in the preceding section, the tools assist in migrating functions to methods. One tricky point remains for the invocation of methods which differ from function invocations in that they need a receiver.

10.3.2 Generalizing library and paradigm use migration with adaptive rules

This section describes five rules for translating native types, primitives, libraries or SDK elements. They can also adapt any usage from procedural to object-oriented, like inferring the receiver according to the available mappings. The rules are: (i) Simple rename, (ii) Rename Map arguments with a static receiver, (iii) Rename Map arguments with this receiver, (iv) Rename Map arguments with argument receiver, (v) Autowrap missing library.

10.3.2.1 Simple rename: Renaming types and simple invocations

- **Description.** According to the available mapping information, this rule renames any type or invocation implying *no paradigm shift*. This rule ensures there are no parameters on the declaration. It contemplates the possibility of the source declaration being addressed with no arguments.
- **Specification.**

  Condition

  1. There is a target declaration mapped **AND**
  2. The target reference can refer to the target declaration (*e.g.* A function invocation cannot refer to a method) **AND**

     (a) Mapped source and target declarations have no parameters **OR**

   (b) All the source parameters are optional **AND** The target reference has no arguments.

Operation

1. Ensure that the target reference uses as identifier the same identifier as the target declaration.
2. Ensure that the target reference refers to the target declaration.

- **Example.** Let us consider the declaration of a variable using a type reference to the primitive type "string". We map the MS Access type string with the Java class String. After translating this variable to Java, the engine would execute this rule: (i) The type reference target can point to the String class. (ii) The type reference has no parameters; therefore, all the conditions are met. The operation will check that the literal hold by the target reference is *String* instead of *string* (mind the uppercase), and therefore modify the type reference to fit in.

10.3.2.2 Rename & Map arguments with *static* receiver

- **Description.** According to the available mapping information, this rule translates any function invocations into method invocations. This rule applies only to mappings between a function and a static method.

  The function invocation is translated as a method invocation using the class defining the mapped method as the expression's receiver. The method invocation is created using the mapped method selector as the invocation selector. The method arguments are rearranged according to the mapping.

- **Specification.**

Condition

1. There is a target declaration mapped **AND**
2. The target reference **cannot** refer to the mapped target declaration **AND**
3. The mapped target declaration is a method or an attribute **AND**
4. The mapped target declaration is *static*.

Operation

1. If the mapped target declaration is an attribute, write an attribute access expression (which is a reference) using the attribute's parent as a receiver.
2. If the mapped target declaration is a method, write a method invocation expression (which is a reference) using the method's parent as a receiver.

3. Set the arguments in the new target reference according to the mapping information.
4. Set the new target reference to the mapped target declaration.

- **Example.**

  Let us consider the mapping between the MS Access function Date and the static method "LocalDate.now". After copying the usage of this function as function invocation in java, the engine would execute this rule: (i) a target declaration is mapped to "Date": "LocalDate.now". (ii) The function invocation cannot refer to it: it is a method. (iii) "LocalDate.now" is static. With all the conditions met: the operation would yield a new expression to replace the old one. This expression would be a method invocation "LocalDate.now()"

10.3.2.3 Rename & Map arguments with *self* or *super* receiver

- **Description.** According to the available mapping information, this rule translates function invocations into method invocations. This rule applies only to mappings between a function and a non-static method and to function invocations to be translated into a method invocation from the same class as the mapped method. The function invocation is translated into a method invocation using either *self/this* or *super* as the receiver of the expression. The receiver is chosen according to whether the method has been overridden. The method invocation is created using the mapped method selector as the invocation selector. The method arguments are rearranged according to the mapping.

- **Specification.**

  Condition

  1. There is a target declaration mapped **AND**
  2. The target reference **cannot** refer to the mapped target declaration **AND**
  3. The mapped target declaration is a method or an attribute **AND**
  4. The target reference is used in the context of a method **AND**
  5. The mapped target declaration is reachable from the calling context by using self or super.

  Operation

  1. If the mapped target declaration is an attribute, write an attribute access expression (which is a reference) using super or self as the receiver.
  2. If the mapped target declaration is a method, write a method invocation expression (which is a reference) using super or self as the receiver.

3. Set the arguments in the new target reference according to the mapping information.
4. Set the new target reference to refer to the mapped target declaration.

- **Example.** For example, invocation of the function *Assert* as used in the example. Let us consider the mapping of the function Assert with the method Asserter.assertEquals. After copying the usage of this function as function invocation in Java inside a method of a subclass of TestCase. the engine would execute this rule: (i) there is a target declaration mapped: Asserter.assertEquals. (ii) The function invocation cannot refer to it: it is a method. (iii) Asserter.assertEquals is reachable by using self. With all the conditions met: the operation would yield a new expression that will replace the old one. This expression would be a method invocation "this.assertEquals(...)"

10.3.2.4 Rename & Map arguments with *argument* receiver

- **Description.** According to the available mapping information, this rule translates function invocations into method invocations. This rule applies only to mappings between a function and a non-static method and with mappings designing an argument as the receiver of the expression. The function invocation is translated into a method invocation using the argument mapped as the receiver of the expression. The method invocation is created using the mapped method selector as the invocation selector. The method arguments are rearranged according to the mapping.

- **Specification.**

  Condition

  1. There is a target declaration mapped **AND**
  2. The target reference **cannot** refer to the mapped target declaration **AND**
  3. The mapped target declaration is a method or an attribute **AND**
  4. The mapping offers a parameter as a receiver. **AND**
  5. The type of receiver can reach the target declaration mapped.

  Operation

  1. If the mapped target declaration is an attribute, write an attribute access expression (which is a reference) using the argument that responds to the mapping as a receiver.
  2. If the mapped target declaration is a method, write a method invocation expression (which is a reference) using the argument that responds to the mapping as a receiver.

3. Set the arguments in the new target reference according to the mapping information.
4. Set the new target reference to refer to the mapped target declaration.

- **Example.** For example, invocation of the function Len as used in the example. We map the function *Len* with the method from *String.length*, to be resolved by the argument.

  After copying the usage of this function as function invocation in java, with a string as an argument. the engine would execute this rule: (i) there is a target declaration mapped: *String.length*. (ii) The function invocation cannot refer to it: it is a method. (iii) *String.length* is reachable by using the argument. With all the conditions met: the operation would yield a new expression that will replace the old one. This expression would be a method invocation "argument.length()"

10.3.2.5 Autowrap

- **Description.** Defines a new declaration that wraps a call to the target declaration respecting the source declaration order of parameters. If a piece of code uses a library artefact we cannot map, we have two options: (i) dismiss the code and (ii) migrate the library manually. If we dismissed the code, we would not be translating this piece of code. We consider then that the only alternative is to migrate the library manually. We propose automatically generating an equivalent structure to let the developer define the behaviour to ease the job.

- **Specification.**

  Condition

  1. There is **no** target declaration mapped **AND**
  2. The source declaration was defined in a library **AND** It is **not** a type.

  Operation

  1. Automatically translates the library component into a pre-established destination.

- **Example.** Let us consider the function `Chr(int)` defined in the module Strings. Let us consider that no map is provided. After copying the usage of this function as function invocation in java, the engine will check this rule. (i) the function has no mapping established. (ii) the function is defined in a library. With all the conditions met: the operation would migrate the function's header into a default target entity as a static method and raise an error

as the default implementation. This automatic migration would establish a mapping which in the next iteration would automatically modify the invocation to `Default.Chr`.

10.3.3 Experimental Artefacts

Our validation is based on the analysis of a simple language migration. In this section, we present our migrating source and target projects. We also analyse the relationship between these projects.

10.3.3.1 Crafting a representative source project

As we said, our project stems from a collaboration with Berger-Levrault, a major IT company. Therefore our validation concerns one of the running migration projects. There are many reasons not to use an industrial project as a source for validation. First, the language translation is a big part of our industrial migration, but it is not all. The details required to grasp the overwhelming complexity of the problem cannot fit into half a page in a validation. Second, it will be an undisclosable source of proprietary code, making the example less interesting.

We propose then to make up our source project. But to make it an interesting case of study, we require two things: (i) it must respond to an actual project's library and runtime usage. (ii) it must be testable.

10.3.3.2 Choice of library elements

To migrate the libraries, we first detect the artefacts used from each library, then determine which are more used in the ePaie system as a prioritisation criterion. We run the case study presented in Section 8.3 to do so.

Demographics. The chosen ePaie system is the Microsoft Access project that consists of **18 subprojects**, summing up **1000 UI widgets** and **700k lines of code** and using **19 different libraries**. This project uses almost **690 different elements** from different libraries.

- **First filtering.** We use the analysis tool proposed in Chapter 7 to select the most meaningful library artefacts by applying the Pareto principle (also known as the 80%-20% rule). That yielded a list of 160 elements that represent 80% of library usage in the project.
- **Second filtering**. From the yielded list, we filtered out all the elements regarding MS Access infrastructure, UI, file system or those which cannot be unit-tested.

  MS Access infrastructure provides many helping elements unconceivable on the target. For example, the "CurrentDB" global. Our language translation

project does not consider UI (which will be migrated differently). Our target is either a back-end or a front-end program. While MS Access makes sense to access the user's files, it is not the same for either back-end or front-end applications. Therefore all the code using those functions must be rewritten. Finally, the ADODB[2] and DAO [3] objects (Recordset, Database and Connection classes) cannot be unit-tested.

We initially did the test and tried to add it to the validation, but it cannot be unified since a single testing method over the database requires some setup and cleanup.

10.3.3.3 Writing tests

We implemented an extremely humble testing library in MS Access. Each test is a function. We have no setUp. We have a single implementation of Assert, which receives two entities to compare for equality and a text to show if the assertion fails.

We created three different modules according to the kind of tested entity: types, functions/subprocedures, and constants/globals.

Testing types: The test asserts the default value of a declared variable. An example is given in Listing 10.8.

```
1 Public Function test_ErrObject()
2    Dim var As ErrObject
3    Call Assert(var Is None, True, "None expected")
4 End Function
```

Listing 10.8: Default value is None.

Testing Constants: The test that asserts the default value. An example is given in Listing 10.9.

```
1 Public Function testVbNullString()
2    Call Assert(vbNullString, "", "Constant Expected")
3 End Function
```

Listing 10.9: Null string is an empty string.

Testing Functions: The test asserts examples of usage. An example is given in Listing 10.10.

```
1 Public Function testUCase()
2    Call Assert(UCase("hellow"), "HELLOW", "Upper case expected")
3 End Function
```

Listing 10.10: Upper case must be upper.

[2]Microsoft ActiveX Data Object Database library
[3]Microsoft Data Access Object library

10.3.3.4 The source project

The source project we propose for the validation consists of **55 tests** split into four different modules. 14 tests of constants and globals, 18 tests of types, 21 tests of functions.

10.3.3.5 Getting to know our targets

The target applications are available in the target folder.

Our validation translates 53 tests, detailed in Section 10.3.3.1, to two different targets. Here we briefly describe what we find in these two target projects.

Decisions in common. This validation is not about migrating tests to be used by a Test framework since we have found none in MS Access, and there is no usage of something similar in our industrial projects. Therefore, to simplify and avoid polluting the engine configuration, we expect the target projects to have four different testing classes to receive the translation of the tests of each of our four MS Access testing modules.

The Java target. Java is a popular statically typed object-oriented language. The target is a Maven project configured to compile Java 1.8 with a single dependency: JUnit3. We choose JUnit3 instead of the latest version to avoid the usage of annotations. Using annotations would force us to implement differently between Java and Pharo.

This dependency is required to support testing. The rest of the available dependencies are the many definitions in the JDK. As previously existing code, we defined four empty test case classes using the Eclipse default template to simplify the experiment. These classes are a subclass of the junit.framework.TestCase class.

The Pharo target. Pharo is a popular open-source dynamically typed pure object-oriented language inspired by Smalltalk. The target is a Pharo package that includes the four expected TestCase classes developed in Pharo 10. Pharo ships many dependencies within the Pharo image, from the definition of Object and SmallInteger to the SUnit testing framework. During our Pharo ASG loading process, we cut down the available dependencies to ensure we do not load useless stuff.

10.3.4 Experimental Context

10.3.4.1 Validation process

Our validation process follows these steps

1. Load the source and target projects.

2. Setup engine rules and mappings.
3. Translate test functions as methods in previously existing test classes.
4. Measure the translated code with the following criteria
   - Is the resulting test Parsable?
   - Is the resulting test Compilable?
   - Is the resulting test error, failure or success?

10.3.4.2 Language migration engine setup

- **Installed Rules.** We used the same kind of translation and adaptive rules for both engines. We use the following five translative rules: CopyAsClass (see Section 10.1.4.1), CopyAsStaticMethod (see Section 9.4.2.1), Copy (see Section 9.5.1.1) and GlobalToAttribute, which transforms all global variables or constants into a static attribute. We need a different rule to transform three binary operations according to the target language. CopyReplaceBinaryOperator, which copies a BinaryOperator structure replacing the operator. There are two ways to express equality in MS Access: "`A Is B`" and "`A = B`". Both are translated as "`A == B`" in Java and as "`A = B`" in Pharo. There is also the concatenation in MS Access: "`A & B`". Translated as "`A + B`" in Java, and "`A , B`" in Pharo. We want to note that this set of rules is only enough to translate our examples, which use few concepts. We use the five rules discussed in Section 10.3.2 for the adaptive rules.
- **Mappings.** We configure the engine mappings according to the target ASG. Table 10.3 lists some of the mapped elements.

10.3.4.3 Research questions

RQ#1 Given the reference of the source program and the declaration that it is expected to refer to the target program Can we automatically propose a pertinent translation?

1. to use the target environment's pre-existing artefacts (SDK, Libraries, Types, etc.)?
2. to use the target environment's paradigm?

RQ#2 Using a heterogenous unified meta-model, does it helps to reuse rules?

RQ#3 Does the approach detach **what**, **how** and **when** to translate code?

Table 10.3: Mapping examples table

VBA	Java	Pharo
VBA.Single	java.lang.Float	Kernel-Numbers.Float
VBA.String	java.lang.String	Collections-Strings.String
VBA.ubyte	java.lang.Byte	Kernel-Numbers.SmallInteger
VBA.uint	java.lang.Integer	Kernel-Numbers.Integer
VBA.uint16	java.lang.Short	Kernel-Numbers.SmallInteger
VBA.uint32	java.lang.Integer	Kernel-Numbers.Integer
VBA.ulong	java.lang.Long	Kernel-Numbers.Integer
VBA.ulonglong	java.lang.Long	Kernel-Numbers.Integer
VBA.USERDEFINED	java.lang.Object	Kernel-Objects.Object
VBA.Variant	java.lang.Object	Kernel-Objects.Object
VBA.VOID	java.lang.Void	None
VBA.Void	java.lang.Void	None
VBA.Win32WideString	java.lang.String	Collections-Strings.String
Win32.Win32Variant	java.lang.Object	Kernel-Objects.Object

10.3.5 Validation

Table 10.4 presents an overview of the testing modules Constants, Functions and Type tests.

Table 10.4: Translation Results.

Test case	Total	Mapped	Parsed	Compiled	Test Result		
					Success	Fail	Error
Constants	14	1	14	1	1	0	0
Functions	21	11	21	16	8	3	5
Type	18	13	18	9	4	4	1
Java Total	53	25	53	26	13	7	6
Constants	14	4	14	14	3	0	11
Functions	21	14	21	21	10	3	8
Type	18	13	18	18	7	10	1
Pharo Total	53	31	53	53	20	13	20

We want to insist that the engine was configured with the same set of translation rules (with a slightly different configuration), and with the same set of adaptive rules to produce two different OOP targets with different types of typing systems and libraries. The main difference between the configurations was the source and target ASG mappings.

- **Mapping Parsing and compiling.** Figure 10.4 shows how many entities

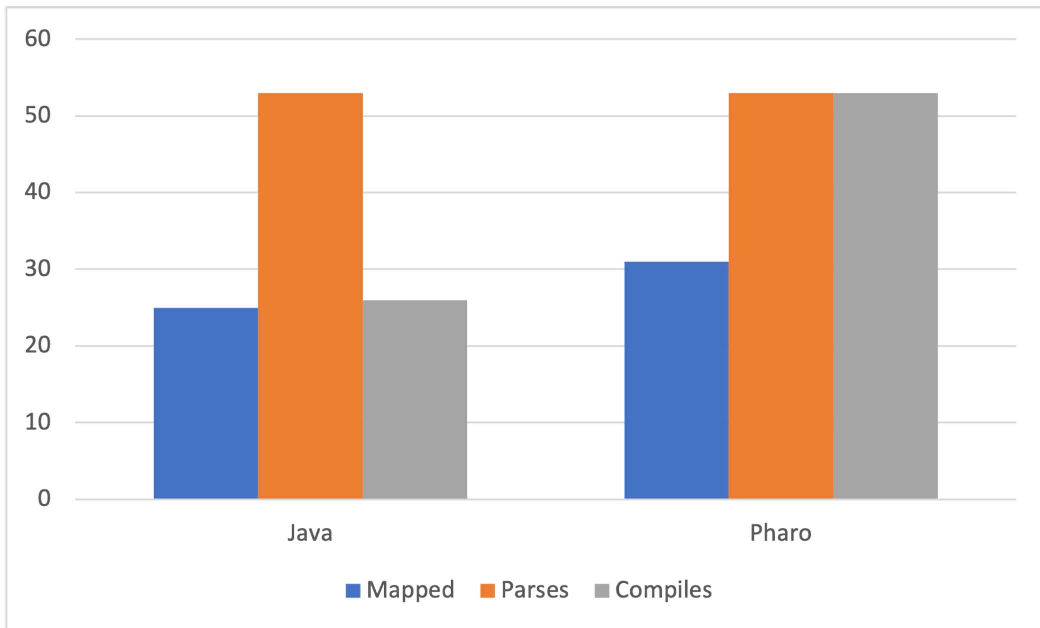


Figure 10.4: Bar chart comparing the number of mappings and the impact on parsing and compiling.

were successfully mapped, how many translated tests were successfully parsed, and how many were correctly compiled. For Java, we mapped 47% of entities. 58% of them in Pharo. For both targets, the produced source code is verified by the SmaCC[4] Java parser and the Pharo parser. The Java compiler compiles 47% of the translated testing methods. The Pharo compiler compiles 100%. The compiling rate seems tightly related to the mapping in the case of Java since the compiling errors come from wrong typing. We argue that most of them stem from the MS Access type named Variant, which we mapped to Object in Java. MS Access Variant accepts any value but keeps the type of the element (object or not). In Java, the type Object accepts only objects (no primitive values allowed), but if a variable is typed as Object, it loses all singularity.

- **Tests.** Figure 10.5 shows in a stacked bar plot the amount of executed tests split by the result: success, failure and error, and at the same time, it splits the errors and failures by kind. All the successes make sense, so we do not classify them. We have three kinds of failures: Only in Java do we find failures because of the wrong literal translation. *i.e.* testing if a double variable value is 0, it must be used 0d. Only in Pharo do we find failures related to untyped variables. In Pharo, all the variables start with nil because all possible values are objects.

[4] https://github.com/j-brant/SmaCC

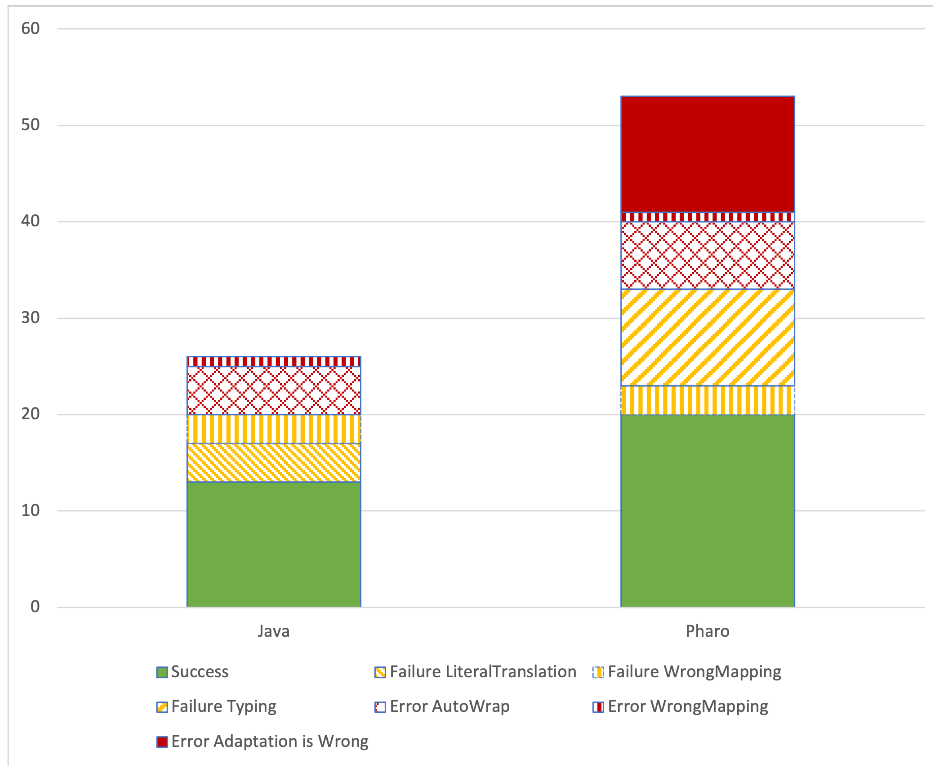


Figure 10.5: Stacked bar chart comparing tests results per language

In both targets, we find failures due to wrong mapping. Some functions are not entirely equivalent in their behaviours. For example, Str(3) is expected to return `"  3"` note the whitespace. We found no method doing the same in Java or Pharo. We have three kinds of errors: We find errors on both targets due to the Autowrap rule's usage. These errors a fully expected since the rule generates methods that raise exceptions asking the developer to implement the behaviour. On both targets, we find errors due to wrong mapping. *i.e.* MS Access allows using strings indicating time to create a Date. Finally, only in Pharo do we find errors because the job of the Autowrap rule was not enough: In Pharo, all the attributes of an object or class are only accessible by the class itself or subclasses. We must implement the accessors when the Autrowrap rule creates equivalent globals (as class-side "static" attributes). And all the accesses to attributes must be rewritten as accessor invocation.

10.3.6 Threats to Validity

There are two main internal threats to the validation proposed.

- **Face validity.** The written tests are based on simple usage of functions.

These tests do not test the boundary cases of the library uses. This means that the proposed mappings may be partially correct.

- **Construct validity.** The second threat is that the validation is done over simple tests. We scoped the problem explicitly to a library and paradigm use migration. There are no control flow statements and no other kind of complex device to ensure that we have the same rules for both targets and that we minimise the noise that could bring the use of other rules to the experiment. However, this also means that the validation does not cover the interrelation of the adaptive rules with many other kinds of rules.

There is also an external threat to validity.

Ecological validity The choice of Junit3 was compulsory. We need a testing framework following the same conventions as SUnit to avoid needing any rule specific to a target. This validation would not apply out of the box to a technology that does not have a testing framework with the same conventions as SUnit.

10.4 Conclusion

In this chapter, we validate our approach over three different experiments and show how the tool can address multiple requirements through a mixture of interactivity and the usage of rules.

In these validations, we exposed the approach to producing Java with Springboot and Typescript with Angular and Pharo. These validations aim to measure and understand the ability of the approach to deal with language, library, paradigm, architecture and GUI migration. On the one hand, the tool lets developers decide manually about the architectural and paradigmatic migration. On the other hand, the tool automates code transformation, translation and generation.

In Section 10.1, we validate that the approach is helpful when it is required to produce partial results, split into multiple targets, translate, generate code out of infrastructure descriptions, and shift paradigms.

In Section 10.2, we validate that the approach can generate models and data access scaffolding from analysing tables.

Finally, in Section 10.3, we validate that the approach is helpful to migrate library uses by simple code rewriting, which also includes a paradigm shift from the point of view of the usage of artefacts by inferring receivers.

Deeper validations must be made to measure the reduced effort, but the results are still encouraging, and we empower the developers allowing them to conduct the migration.

CHAPTER 11

Conclusion

Contents

11.1 Migrating Microsoft Access applications 180
11.2 Problems and Contributions 180
11.3 Meeting the requirements 182
11.4 Future work 185

In this chapter we conclude the work of the thesis. For doing so, in Section 11.1, we brief the contributions of the manuscript.

In Section 11.2, we go back to the different families of problems this thesis addresses: modelling, understanding and transforming (presented in Chapter 1) and relate them with each chapter.

In Section 11.3, we go back to the requirements listed in Chapter 3 and explain how our approach solves them one by one, finishing with a table that compares our approach with the approaches presented in Chapter 2.

We close the thesis in Section 11.4, where we present future works.

11.1 Migrating Microsoft Access applications

The literature witnesses that any migration from a library update to a language migration are complex endeavour. It is known that migrating from a rich language to a general-purpose language is remarkably complex [Brant 2010]. Migrating 4GL systems to general-purpose languages is highly complex since 4GL languages are rich in grammatical constructs.

Migrating these systems to web technologies has an extra complexity: 4GL systems mixes concerns that in the target technologies are expected to be solved by at least two different applications, written in two different languages, running in two different environments, with different libraries. This makes the migration include application splitting and architectural migration.

To address this industrial migration, we studied Microsoft Access in-depth, learning how to reverse engineer such projects. We also studied our technological targets: Java with SpringBoot for the micro-services-based back-end and Typescript with Angular for the micro-application-based front-end.

To understand the different kinds of tasks entailing an industrial migration, we studied multiple processes for software migration and devised our own process. Moreover, we tooled this process by providing:

A tool to *understand software migration* measuring the technological distance, visualising the architectural target violations and measuring the progress of the migration.

A tool to *conduct a software migration* with multiple aspects as ours, we implement an engine allowing us to produce partial migration and to split pieces of the source system into any of the multiple targets. We devise an interactive approach allowing the developer to make architectural and design decisions during the migration, keeping a degree of automation to reduce the effort required.

11.2 Problems and Contributions

As we said in Chapter 1. Semi-automatically migrating large Microsoft Access applications to multiple targets face three families of problems.

(1) Modelling: Any (semi-) automatic approach requires a model. Moreover, the model must support representing multiple technologies and be extracted automatically from the source code.

(2) Understanding: We do not directly address planning, but we do address the understanding of the source application and how it is affected by the chosen targets, with different degrees of detail that can be used in both planning (a strategic step) and understanding (an operational step).

(3) Split and transform: the approach must propose a way to split an application by concern and transform it into multiple technological targets, for which the approach must be technologically agnostic.

11.2.1 Chapters and article contributions to the problems.

We now list all the chapters from Chapter 5 to Chapter 10, relating them to the family of problems they address.

- Chapter 5 contributes to “(2) understanding the source system” by understanding the limitations of the source system’s technology and “(1) modelling” by exploring the threats to reverse engineering inherent to Microsoft Access technology and how we make a model out of binary representation.

- Chapter 6 contributes to “(1) modelling” by applications migration using the Heterogenous Unified Meta-Model.

- Chapter 7 contributes to “(2) understanding the distance between the source system and target technology” by measuring the technological gap between them based on metrics and learning about the architectural incompatibility using visualisations. This understanding helps not only to get insight into the size and complexity of the task but also to gain insight into the requirements of the tasks related to “(3) split and transform”.

- Chapter 8 contributes to the “(2) understanding of three industrial studies” by using our metrics and visualisations: the risk and complexity assessment for the software migration of (i) eGRC system and (ii) CyclePaie project, and (iii) the task selection and prioritisation process of library migration of the ePaie system. This information is valuable for developing transforming rules, contributing to “(3) split and transform”.

- Chapter 9 contributes to “(3) split and transform” by providing an Interactive Iterative/Incremental, Tooled, Rule-Based approach to Software Migration, implemented over a Context-Aware Partial Translation engine based on immediate and delayed Rule application. The interactive approach contributes to “(2) understanding the target”.

- Chapter 10 contributes to “(3) split and transform” by validating the approach from three different perspectives: (i) The validation of the iterative, interactive approach to migrating an MS Access form to the front and back end; (ii) The validation of the iterative, interactive approach to migrating MS Access tables and queries to the front and back end and (iii) the validation of the iterative, interactive approach to migrating MS Access library and procedural-paradigm uses to Pharo and Java.

11.3 Meeting the requirements

Back in Chapter 3, we present multiple requirements of our approach.

Agnostic: Our approach is based on modelling source and target systems. The HUMM meta-model used for these systems (presented in Chapter 6) simplifies the definition of reusable rules: such is the case of the AnyCopy rule, and many others presented in Chapter 10. Two artefacts are required for each technology: a model extractor and a code generator. The extractor produces a HUMM out of a system, and the generator produces or modifies a system out of a HUMM.

Programming language migration:

Transform. Our approach implements language transformations based on productive rules. For example, the rule transforming Functions into static Methods is presented in Chapter 9.

Import. Our model (presented in Chapter 6) is an ASG. This kind of structure encodes references. By analysing these references, we can infer all the required imports.

Library and infrastructure migration:

Replace infrastructure. Our approach migrates infrastructure in two ways. One is a particular case of library replacement (see next), and the other way is by using productive rules which extract knowledge and produce the required code (see Chapter 10).

Replace Libraries. Our approach migrates libraries by mapping and applying adaptive rules, as presented in Chapter 9. More examples in Chapter 10.

Paradigm migration:

Procedural. Procedural to object-oriented migration has two parts: the migration of the structures and the migration of the usage. Our approach is interactive; the developer can manually choose how to transform functions into methods, giving him this responsibility. On the other hand, our approach allows decision consistency (as discussed in Chapter 9) by using adaptive rules.

Access First-Class Citizen. The other first-class citizens are transformed into the target by specialised productive rules. In Chapter 10, we analyse and validate the cases of forms, tables and queries.

Architectural migration:

Partial. Our engine is designed (see Chapter 9) to respond to incremental migration, which enables the interactive tool to choose **what** to migrate, in which target.

Split. The interactive aspect of our approach enables the user to split the application by choosing not only what to migrate but also **where** to migrate (see Chapter 9).

Communication. The mapping between models can be used to map elements between source and target systems and between systems. In Chapter 10, we explore this option when migrating an MS Access form and using productive rules and mappings to automate establishing connections between the front and back ends.

GUI migration:

Visual. To migrate the visual aspect of our forms into Angular widgets, we propose using the Casino knowledge model, used for visual migrations within the scope of a rule. We present this rule in Chapter 10.

Behaviour. To migrate behavioural aspects of the forms, we also migrate the event handling functions. We extended the knowledge model to relate these event handlers with the different visual controls of the target widget. We present these rules in Chapter 10.

Developer:

Expertise. To migrate the knowledge of the developers, we develop rules that keep the same naming as in the source systems. We also provide an interactive approach driven by the development team, which helps to capitalise on the knowledge of the target system. (see Chapter 9)

Etiquette. We develop rules that transform the code using the proper target structures to respect the target system etiquette. (see Chapter 9)

Table 11.1: This table lists the requirements proposed in Chapter 3. The first shows the information related to the approach presented in this thesis.

Requirements / Article ID	Agnostic	Lanugage		Library / infrastructure		Paradigm		Architectural			GUI		Developer	
		Translate / Transform	Imports	Infrastructure	Library	Procedural	Access FCC	Partial	Split	Connection	Visual	Behavoral	Keep Names	Etiquette
- Our approach	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1 [Verhaeghe 2019]	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓
2 [Angulo 2018]	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
3 [Garcés 2017]	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✓	✓
5 [Jain 2015]	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗
7 [Ahmad 2014]	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
8 [Bergmayr 2013]	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗
12 [Brant 2010]	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗
15 [De Lucia 2008]	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗
16 [Zhou 2007]	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
17 [Di Santo 2007]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗
19 [Zhang 2004]	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
20 [de Souza 2003]	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗
21 [Martin 2002a]	✗	✓	✗	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗
22 [Zou 2001]	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗
24 [Kontogiannis 1998]	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
30 [Moore 1994]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗

11.4 Future work

Many possibilities come at hand with the end of this phase.

Modelling. We consider that more research has to be done around the concepts when modelling migrating systems. In our work, we proposed to split grammatical concepts which belong to the structure of the language. We find that migrating binary operations are trivial this way. Still, we cannot define a clear cut to define when to use grammatical concepts and when not to use them. More research has to be done to measure the reduction of efforts derived from this decision.

Research must be done to ensure the correctness of models. While our HUMM allows diversity within a single model, it cannot ensure any degree of correctness.

Understanding. More research is required to understand the usability and utility of the proposed metrics and visualisations. Currently, we, the researchers in this project, always use the tools to produce visualisations, metrics and reports used by managers and key users. We want to explore what other uses and interpretations of these metrics and visualisations are possible.

Moreover, we need more research to correlate the complexity measures with the migration's actual complexity to understand the value of this information.

Transforming The interactive nature of our project threats the reproducibility of migration. The order of the steps a tool user takes may change the outcome; applying the same "migration steps" again over the same source system is not trivial: We want to research transforming user interactions into records that can be used to generate a script to reproduce the same behaviour produced by the users.

We want to research the design of rules towards reusability.

Bibliography

[] *OMG Model Driven Architecture*. http://www.omg.org/mda/.

[Abdeen 2011] Hani Abdeen, Stéphane Ducasse and Houari A. Sahraoui. *Modularization Metrics: Assessing Package Organization in Legacy Large Object-Oriented Software*. Rapport technique, RMod – INRIA Lille-Nord Europe, 2011.

[Agouf 2022] Nour Jihene Agouf, Stéphane Ducasse, Anne Etien and Michele Lanza. *A New Generation of Class Blueprint*. In 2022 Working Conference on Software Visualization (VISSOFT), pages 29–39. IEEE, 2022.

[Agrawal 2003] Aditya Agrawal, Gabor Karsai and Feng Shi. *A UML-based graph transformation approach for implementing domain-specific model transformations*. International Journal on Software and Systems Modeling, pages 1–19, 2003.

[Ahmad 2014] Aakash Ahmad and Muhammad Ali Babar. *A Framework for Architecture-Driven Migration of Legacy Systems to Cloud-Enabled Software*. In Proceedings of the WICSA 2014 Companion Volume, WICSA '14 Companion, New York, NY, USA, 2014. Association for Computing Machinery.

[Aho 1972] Alfred V. Aho and Jeffrey D. Ullman. The theory of parsing, translation and compiling volume I: Parsing. Prentice-Hall, 1972.

[Aho 1986] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. Compilers: Principles, techniques and tools. Addison Wesley, Reading, Mass., 1986.

[Allen 2001] E. Allen and T. Khoshgoftaar. *Measuring Coupling and Cohesion of Software Modules: An Information Theory Approach*. In Seventh International Software Metrics Symposium, 2001.

[Allier 2011] Simon Allier, Salah Sadou, Houari Sahraoui and Régis Fleurquin. *From object-oriented applications to component-oriented applications via component-oriented architecture*. In 2011 Ninth Working IEEE/IFIP Conference on Software Architecture, pages 214–223. IEEE, 2011.

[Amokrane 2020] Nawel Amokrane, Jannik Laval, Philippe Lanco, Mustapha Derras and Nejib Moala. *Analysis of Data Exchanges, Towards a Tooled Approach for Data Interoperability Assessment*. In Intelligent Systems: Theory, Research and Innovation in Applications, pages 345–363. Springer, 2020.

[Angulo 2018] Guisella Angulo, Daniel San Martín, Bruno Santos, Fabiano Cutigi Ferrari and Valter Vieira de Camargo. *An Approach for Creating KDM2PSM Transformation Engines in ADM Context: The RUTE-K2J Case*. In Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse, SBCARS '18, pages 92–101, New York, NY, USA, 2018. Association for Computing Machinery.

[Anquetil 2013] Nicolas Anquetil and André Hora. *Assessing the Quality of Architectural Design Quality Metrics*. Rapport technique, Inria Lille Nord Europe, 2013.

[Anquetil 2020] Nicolas Anquetil, Anne Etien, Mahugnon Honoré Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatija Djareddir, Jèrôme Sudich and Mustapha Derras. *Modular Moose: A new generation of software reengineering platform*. In International Conference on Software and Systems Reuse (ICSR'20), numéro 12541 de LNCS, December 2020.

[Appel 2002] Andrew W. Appel. Modern compiler implementation in Java. Cambridge University Press, New York, NY, USA, second édition, 2002. with Jens Palsberg.

[Aranega 2009] Vincent Aranega, Jean-Marie Mottu, Anne Etien and Jean-Luc Dekeyser. *Traceability Mechanism for Error Localization in Model Transformation.* In ICSOFT (1), pages 66–73, 2009.

[Aranega 2014] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry and Jean-Luc Dekeyser. *Towards an automation of the mutation analysis dedicated to model transformation*. Software Testing, Verification and Reliability, 2014.

[Arcelli 2008] Francesca Arcelli, Christian Tosi and Marco Zanoni. *Can Design Pattern Detection Be Useful for Legacy Systemmigration towards SOA?* In Proceedings of the 2nd International Workshop on Systems Development in SOA Environments, SDSOA '08, page 63 to 68, New York, NY, USA, 2008. Association for Computing Machinery.

[Artho 2001] Cyrille Artho and Armin Biere. *Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs*. In Proceedings of the 13th Australian Conference on Software Engineering, pages 68–, Washington, DC, USA, 2001. IEEE Computer Society.

[Aruna 2008] M. Aruna, M.P. Suguna Devi and M. Deepa. *Measuring the Quality of Software Modularization Using Coupling-Based Structural Metrics*

for an OOS System. In ICETET '08: Proceedings of the 2008 First International Conference on Emerging Trends in Engineering and Technology, pages 1130–1135, Washington, DC, USA, 2008. IEEE Computer Society.

[Bacon 1996] David F. Bacon and Peter F. Sweeney. *Fast Static Analysis of C++ Virtual Function Calls*. In Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.

[Balaban 2005] Ittai Balaban, Frank Tip and Robert Fuhrer. *Refactoring support for class library migration*. ACM SIGPLAN Notices, vol. 40, no. 10, pages 265–279, 2005.

[Ball 1999] Thomas Ball. *The Concept of Dynamic Analysis*. In Proceedings of the European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC'99), numéro 1687 de LNCS, pages 216–234, Heidelberg, sep 1999. Springer Verlag.

[Balmas 2009] Françoise Balmas, Alexandre Bergel, Simon Denier, Stéphane Ducasse, Jannik Laval, Karine Mordal-Manet, Hani Abdeen and Fabrice Bellingard. *Software metric for Java and C++ practices (Squale Deliverable 1.1)*. Rapport technique, INRIA Lille Nord Europe, 2009.

[Bansiya 1999] Jagdish Bansiya, Letha Etzkorn, Carl Davis and Wei Li. *A Class Cohesion Metric for Object-Oriented Designs*. Journal of Object-Oriented Programming, vol. 11, no. 8, pages 47–52, January 1999.

[Bart Du Bois 2006] Bart Du Bois, Jan Verelst, Jan Verelst and Marijn Temmerman. *Does God Class Decomposition Affect Comprehensibility?* In Proceedings of the IASTED International Conference on Software Engineering, Innsbruck, Austria, 2006. IASTED/ACTA Press.

[Basili 1995] Victor R. Basili, Lionel Briand and Walcélio L. Melo. *A Validation Of Object-Oriented Design Metrics As Quality Indicators*. IEEE Transactions on Software Engineering, pages 751–761, 1995.

[Baxter 1998] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna and Lorraine Bier. *Clone Detection Using Abstract Syntax Trees*. In Proceedings of the International Conference on Software Maintenance (ICSM 1998), pages 368–377. IEEE Computer Society, Washington, DC, USA, 1998.

[Bergmann 2012] Gábor Bergmann, István Ráth, Gergely Varró and Dániel Varró. *Change-driven model transformations: Change (in) the rule to rule the change*. Software & Systems Modeling, vol. 11, pages 431–461, 2012.

[Bergmayr 2013] Alexander Bergmayr, Hugo Bruneliere, Javier Luis Canovas Izquierdo, Jesus Gorronogoitia, George Kousiouris, Dimosthenis Kyriazis, Philip Langer, Andreas Menychtas, Leire Orue-Echevarria, Clara Pezuela*et al. Migrating legacy software to the cloud with ARTIST*. In 2013 17th European Conference on Software Maintenance and Reengineering, pages 465–468. IEEE, 2013.

[Beynon-Davies 1999] P Beynon-Davies, C Carne, H Mackay and D Tudhope. *Rapid application development (RAD): an empirical review*. European Journal of Information Systems, vol. 8, no. 3, pages 211–223, 1999.

[Bianchi 2003] Alessandro Bianchi, Danilo Caivano, Vittorio Marengo and Giuseppe Visaggio. *Iterative reengineering of legacy systems*. IEEE Transactions on Software Engineering, vol. 29, no. 3, pages 225–241, 2003.

[Bieman 1994] J.M. Bieman and L.M.Ott. *Measuring Functional Cohesion*. IEEE Transactions on Software Engineering, vol. 20, no. 8, pages 644–658, August 1994.

[Binkley 1998] Aaron B. Binkley and Stephen R. Schach. *Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures*. In ICSE '98: Proceedings of the 20th international conference on Software engineering, pages 452–455, Washington, DC, USA, 1998. IEEE Computer Society.

[Blondeau 2015] Vincent Blondeau, Nicolas Anquetil, Stéphane Ducasse, Sylvain Cresson and Pascal Croisy. *Software metrics to predict the health of a project?* In International Workshop on Smalltalk Technologies IWST'15, Brescia, Italy, July 2015.

[Bragagnolo a] Santiago Bragagnolo, Ducasse Stéphane, Anquetil Nicolas and Mustapha Derras. *Interactive, Iterative, Tooled, Rule-Based Migration of Microsoft Access to Web Technologies*. IN SUBMISSION.

[Bragagnolo b] Santiago Bragagnolo, Ducasse Stéphane, Anquetil Nicolas and Mustapha Derras. *Understanding the Migration of Applications with Typing Ontologies*. IN SUBMISSION.

[Bragagnolo 2020a] Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse, Seriai Abderrahmane and Mustapha Derras. *Analysing Microsoft Access Projects: Building a model in a Partially Observable Domain*. In International Conference on Software and Systems Reuse (ICSR'20), numéro 12541 de LNCS, December 2020.

[Bragagnolo 2020b] Santiago Bragagnolo, Benoît Verhaeghe, Abderrahmane Seriai, Mustapha Derras and Anne Etien. *Challenges for Layout Validation:*

Lessons Learned. In International Conference on the Quality of Information and Communications Technology, QUATIC'2020, September 2020.

[Bragagnolo 2021a] Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse, Abderrahmane Seriai and Derras Mustapha. *Software Migration: A Theoretical Framework (A Grounded Theory approach on Systematic Literature Review)*. Rapport technique, Berger-Levrault and Inria Lille Nord Europe, 2021.

[Bragagnolo 2021b] Santiago Bragagnolo, Abderrahmane Seriai, Stéphane Ducasse and Mustapha Derras. *Risk and Complexity Assessment on the Context of Language Migration*. In International Conference on the Quality of Information and Communications Technology, QUATIC'2021, September 2021.

[Bragagnolo 2022] Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse and Derras Mustapha. *Reporting Context Aware Partial Translation engine based on immediate and delayed Rule application*. Rapport technique, Inria, December 2022.

[Bragagnolo 2023] Santiago Bragagnolo, Ducasse Stéphane, Anquetil Nicolas, Abderrahmane Seriai and Mustapha Derras. *Alce: Predicting Software Migration*. Rapport technique, Inria, January 2023.

[Brant 1998] John Brant and Don Roberts. *"Good Enough" Analysis for Refactoring*. In Object-Oriented Technology Ecoop '98 Workshop Reader, LNCS, pages 81–82. Springer-Verlag, 1998.

[Brant 2010] John Brant, Don Roberts, Bill Plendl and Jeff Prince. *Extreme Maintenance: Transforming Delphi into C#*. In ICSM'10, 2010.

[Briand 1997] Lionel Briand, Prem Devanbu and Walcelio Melo. *An investigation into coupling measures for C++*. In ICSE '97: Proceedings of the 19th international conference on Software engineering, pages 412–421, New York, NY, USA, 1997. ACM.

[Brodie 1995] Michael L. Brodie and Michael Stonebraker. Migrating legacy systems. Morgan Kaufmann, 1995.

[Chikofsky 1990] Elliot Chikofsky and James Cross II. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, vol. 7, no. 1, pages 13–17, January 1990.

[Chisnall 2013] David Chisnall. *The challenge of cross-language interoperability*. Communications of the ACM, vol. 56, no. 12, pages 50–56, 2013.

[Cloutier 2016] Jonathan Cloutier, Segla Kpodjedo and Ghizlane El Boussaidi. *WAVI: A reverse engineering tool for web applications*. In 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pages 1–3. IEEE, 2016.

[Comella-Dorda 2000] Santiago Comella-Dorda, Kurt Wallnau, Robert C Seacord and John Robert. *A survey of legacy system modernization approaches*. Rapport technique, Carnegie-Mellon univ pittsburgh pa Software engineering inst, 2000.

[Cornelissen 2008] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie Van Deursen and Jarke J Van Wijk. *Execution trace analysis through massive sequence and circular bundle views*. Journal of Systems and Software, vol. 81, no. 12, pages 2252–2268, 2008.

[Cossette 2012] Bradley E. Cossette and Robert J. Walker. *Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries*. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pages 55:1–55:11, New York, NY, USA, 2012. ACM.

[Czarnecki 2006] Krzysztof Czarnecki and Simon Helsen. *Feature-based survey of model transformation approaches*. IBM systems journal, vol. 45, no. 3, pages 621–645, 2006.

[D'Ambros 2006] Marco D'Ambros and Michele Lanza. *Software Bugs and Evolution: A Visual Approach to Uncover Their Relationship*. In European Conference on Software Maintenance and Reengineering (CSMR), pages 227 – 236. IEEE Computer Society Press, 2006.

[D'Ambros 2007] Marco D'Ambros and Michele Lanza. *BugCrawler: Visualizing Evolving Software Systems*. In Proceedings of the 11th IEEE European Conference on Software Maintenance and Reengineering, CSMR'07, page to be published, 2007.

[D'Ambros 2009] Marco D'Ambros, Michele Lanza and Romain Robbes. *On the Relationship Between Change Coupling and Software Defects*. In Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009), pages 135–144, 2009.

[Davidson 1980] Jack W Davidson and Christopher W Fraser. *The design and application of a retargetable peephole optimizer*. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 2, no. 2, pages 191–202, 1980.

[De Lucia 2008] Andrea De Lucia, Rita Francese, Giuseppe Scanniello and Genoveffa Tortora. *Developing legacy system migration methods and tools for technology transfer*. Software: Practice and Experience, vol. 38, no. 13, pages 1333–1364, 2008.

[De Pauw 2002] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides and Jeaha Yang. *Visualizing the Execution of Java Programs*. In Revised Lectures on Software Visualization, International Seminar, pages 151–162, London, UK, 2002. Springer-Verlag.

[de Souza 2003] Phillip de Souza, Andrew McNair and Jens H Jahnke. *Network-centric migration of embedded control software: a case study*. In Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research, pages 54–65, 2003.

[DeLine 1997] Robert DeLine, Gregory Zelesnik and Mary Shaw. *Lessons on Converting Batch Systems to Support Interaction: Experience Report*. In Proceedings of the 19th International Conference on Software Engineering, ICSE '97, page 195 to 204, New York, NY, USA, 1997. Association for Computing Machinery.

[Demeyer 1999a] Serge Demeyer and Stéphane Ducasse. *Metrics, Do They Really Help?* In Jacques Malenfant, editeur, Proceedings of Languages et Modèles à Objets (LMO'99), pages 69–82. HERMES Science Publications, Paris, 1999.

[Demeyer 1999b] Serge Demeyer, Stéphane Ducasse and Sander Tichelaar. *Why Unified is not Universal. UML Shortcomings for Coping with Round-trip Engineering*. In Bernhard Rumpe, editeur, Proceedings of the International Conference on The Unified Modeling Language (UML'99), volume 1723 of *LNCS*, pages 630–644, Kaiserslautern, Germany, October 1999. Springer-Verlag.

[Demeyer 2001] Serge Demeyer, Sander Tichelaar and Stéphane Ducasse. *FAMIX 2.1 — The FAMOOS Information Exchange Model*. Rapport technique, University of Bern, 2001.

[Demeyer 2002] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz. Object-oriented reengineering patterns. Morgan Kaufmann, 2002.

[Di Santo 2007] Giuseppe Di Santo and Eugenio Zimeo. *Reversing GUIs to XIML Descriptions for the Adaptation to Heterogeneous Devices*. In Proceedings of the 2007 ACM Symposium on Applied Computing, SAC '07, page 1456 to 1460, New York, NY, USA, 2007. Association for Computing Machinery.

[Dig 2005] Daniel Dig and Ralph Johnson. *The Role of Refactorings in API Evolution*. In Proceedings of 21st International Conference on Software Maintenance (ICSM 2005), pages 389–398, September 2005.

[Dong 2007] Xinyi Dong and M.W. Godfrey. *System-level Usage Dependency Analysis of Object-Oriented Systems*. In ICSM 2007. IEEE Comp. Society, 2007.

[Ducasse 1999a] Stéphane Ducasse and Serge Demeyer, editeurs. The FAMOOS object-oriented reengineering handbook. University of Bern, October 1999.

[Ducasse 1999b] Stéphane Ducasse, Matthias Rieger and Serge Demeyer. *A Language Independent Approach for Detecting Duplicated Code*. In Hongji Yang and Lee White, editeurs, Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99), pages 109–118. IEEE Computer Society, September 1999.

[Ducasse 2001a] Stéphane Ducasse. *Reengineering Object-Oriented Applications*. Rapport technique, Université Pierre et Marie Curie (Paris 6), September 2001. TR University of Bern, Institute of Computer Science and Applied Mathematics — iam-03-008.

[Ducasse 2001b] Stéphane Ducasse and Michele Lanza. *Towards a Methodology for the Understanding of Object-Oriented Systems*. Technique et science informatiques, vol. 20, no. 4, pages 539–566, 2001.

[Ducasse 2004] Stéphane Ducasse, Michele Lanza and Roland Bertuli. *High-Level Polymetric Views of Condensed Run-Time Information*. In Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04), pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[Ducasse 2005] Stéphane Ducasse and Michele Lanza. *The Class Blueprint: Visually Supporting the Understanding of Classes*. Transactions on Software Engineering (TSE), vol. 31, no. 1, pages 75–90, January 2005.

[Ducasse 2006] Stéphane Ducasse, Tudor Gîrba and Adrian Kuhn. *Distribution Map*. In Proceedings of 22nd IEEE International Conference on Software Maintenance, ICSM'06, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[Ducasse 2022] Stéphane Ducasse, Guillermo Polito, Oleksandr Zaitsev, Marcus Denker and Pablo Tesone. *Deprewriter: On the fly rewriting method deprecations*. Journal of Object Technologies (JOT), vol. 21, no. 1, 2022.

[Duddy 2004] K Duddy, Anna Gerber, Michael Lawley, Kerry Raymond and Jim Steel. *Declarative transformation for object-oriented models*. Transformation of Knowledge, Information, and Data: Theory and Applications, jan 2004.

[El-Ramly 2006] Mohammad El-Ramly, Rihab Eltayeb and Hisham A Alla. *An experiment in automatic conversion of legacy Java programs to C*. In IEEE International Conference on Computer Systems and Applications, 2006., pages 1037–1045. IEEE, 2006.

[Emerson 1984] Thomas Emerson. *A Discriminant Metric for Module Cohesion*. In Proceedings of the 7th International Conference on Software Engineering (ICSE), 1984.

[Esbai 2015] Redouane Esbai and Mohammed Erramdani. *Model-to-model transformation in approach by modeling: From UML model to Model-View-Presenter and Dependency Injection patterns*. In 2015 5th World Congress on Information and Communication Technologies (WICT), pages 1–6. IEEE, 2015.

[Etien 2010] Anne Etien, Alexis Muller, Thomas Legrand and Xavier Blanc. *Combining independent model transformations*. In Proceedings of the 2010 ACM Symposium on Applied Computing, pages 2237–2243, 2010.

[Etien 2015] Anne Etien, Alexis Muller, Thomas Legrand and Richard F. Paige. *Localized model transformations for building large-scale transformations*. Software & Systems Modeling, vol. 14, no. 3, pages 1189–1213, 2015.

[Feijs 1998] Loe Feijs and Roel De Jong. *3D visualization of software architectures*. Communication of the ACM, vol. 41, no. 12, pages 73–78, 1998.

[Fleurey 2007] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas and Jean-Marc Jezéquel. *Model-Driven Engineering for Software Migration in a Large Industrial Context*. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt and Frank Weil, editeurs, Model Driven Engineering Languages and Systems, volume 4735, pages 482–497, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Fowler 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts. Refactoring: Improving the design of existing code. Addison Wesley, 1999.

[Fox 1997] Geoffrey Fox, Xiaoming Li, Zheng Qiang and Wu Zhigang. *A prototype of Fortran-to-Java converter*. Concurrency: Practice and Experience, vol. 9, no. 11, pages 1047–1061, November 1997.

[Francesca 2008] Arcelli Fontana Francesca, Perin Fabrizio, Raibulet Claudia and Ravani Stefano. *Behavioural Design Pattern detection through dynamic analysis*. In Proceedings of 4th PCODA at the 15th Working Conference on Reverse Engineering (WCRE 2008), pages 11–16, 2008.

[Ganesan 2016] A. Sivagnana Ganesan and T. Chithralekha. *A Survey on Survey of Migration of Legacy Systems*. In Proceedings of the International Conference on Informatics and Analytics, ICIA-16, New York, NY, USA, 2016. Association for Computing Machinery.

[Garcés 2017] Kelly Garcés, Rubby Casallas, Camilo Álvarez, Edgar Sandoval, Alejandro Salamanca, Fredy Viera, Fabián Melo and Juan Manuel Soto. *White-box modernization of legacy applications: The oracle forms case study*. Computer Standards & Interfaces, pages 110–122, October 2017.

[Giese 2010] Holger Giese, Stephan Hildebrandt and Stefan Neumann. Model synchronization at work: Keeping sysml and autosar models consistent, pages 555–579. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Gotti 2016] Zineb Gotti and Samir Mbarki. *Java Swing Modernization Approach - Complete Abstract Representation based on Static and Dynamic Analysis:*. In Proceedings of the 11th International Joint Conference on Software Technologies, pages 210–219. SCITEPRESS - Science and Technology Publications, 2016.

[Govin 2016] Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *How Can We Help Software Rearchitecting Efforts ? Study of an Industrial Case*. In Proceedings of the International Conference on Software Maintenance and Evolution, (Industrial Track), Raleigh, USA, October 2016.

[Govin 2017] Brice Govin, Nicolas Anquetil, Anne Etien, Stéphane Ducasse and Arnaud Monegier Du Sorbier. *Managing an Industrial Software Rearchitecting Project With Source Code Labelling*. In Complex Systems Design & Management conference (CSD&M), Paris, France, December 2017.

[Greevy 2005a] Orla Greevy, Abdelwahab Hamou-Lhadj and Andy Zaidman. *Workshop on Program Comprehension through Dynamic Analysis (PCODA)*. In 12th Working Conference on Software Maintenance and Reengineering (WCRE 2005), pages 232–232, September 2005.

[Greevy 2005b] Orla Greevy, Michele Lanza and Christoph Wysseier. *Visualizing Feature Interaction in 3-D*. In Proceedings of VISSOFT 2005 (3th IEEE International Workshop on Visualizing Software for Understanding), pages 114–119, September 2005.

[Grosse-Kunstleve 2012] Ralf W. Grosse-Kunstleve, Thomas C. Terwilliger, Nicholas K. Sauter and Paul D. Adams. *Automatic Fortran to C++ conversion with FABLE*. Source Code for Biology and Medicine, vol. 7, no. 1, page 5, may 2012.

[Group 1997] Object Management Group. *Meta Object Facility (MOF) Specification*. Rapport technique ad/97-08-14, Object Management Group, September 1997.

[Group 2000] Object Management Group. *Meta Object Facility (MOF) Specification (version 1.3)*. Rapport technique, Object Management Group, March 2000.

[Gunter 1996] Carl Gunter, John Mitchell and David Notkin. *Strategic Directions in Software Engineering and Programming Languages*. ACM Comput. Surv., vol. 28, no. 4, page 727 to 737, December 1996.

[Gustafsson 2000] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Mälardalen University, Sweden and Upssala University, Sweden, May 2000.

[Halstead 1977] Maurice H. Halstead. Elements of Software Science. Elsevier North-Holland, 1977.

[Hayakawa 2012] Tomokazu Hayakawa, Shinya Hasegawa, Shota Yoshika and Teruo Hikita. *Maintaining Web Applications by Translating Among Different RIA Technologies*. GSTF Journal on Computing, page 7, 2012.

[Healey 1992] C. G. Healey. Visualization of multivariate data using preattentive processing. Master's thesis, Department of Computer Science, University of Bristish Columbia, 1992.

[Ira Baxter 1997] Chris Verhoef Ira Baxter Alex Quilici, editeur. Fourth working conference on reverse engineering. IEEE, 1997.

[ISO 2001] ISO. *International Standard – ISO/IEC 9126-1:2001 – Software engineering – Product quality*. Rapport technique, ISO, 2001.

[ISO 2006] ISO. *International Standard – ISO/IEC 14764 IEEE Std 14764-2006*. Rapport technique, ISO, 2006.

[ISO 2011a] ISO. *International Standard – ISO/IEC 25010:2011 – Software engineering – Product quality*. Rapport technique, ISO, 2011.

[ISO 2011b] ISO. *ISO/IEC/IEEE Systems and software engineering – Architecture description*. ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000), pages 1–46, 2011.

[ISO 2015] ISO. *International Standard – ISO/ICE 90003:2018 – Software engineering – Product quality*. Rapport technique, ISO, 2015.

[J. Eder 1994] M. Schrefl J. Eder G. Kappel. *Coupling and cohesion in object-oriented systems*. Rapport technique, Department of Information Systems, University of Linz, Austria, 1994.

[Jain 2015] Suman Jain and Inderveer Chana. *Modernization of Legacy Systems: A Generalised Roadmap*. In Proceedings of the Sixth International Conference on Computer and Communication Technology 2015, ICCCT '15, page 62 to 67, New York, NY, USA, 2015. Association for Computing Machinery.

[Jouault 2010] Frédéric Jouault and Massimo Tisi. *Towards incremental execution of ATL transformations*. In Theory and Practice of Model Transformations: Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings 3, pages 123–137. Springer, 2010.

[Kan 2006] Stephen H. Kan. Metrics and models in software quality engineering. O'Reilly, 2006.

[Kent 2002] Stuart Kent. *Model driven engineering*. In Integrated formal methods, pages 286–298. Springer, 2002.

[Khadka 2014] Ravi Khadka, Belfrit V Batlajery, Amir M Saeidi, Slinger Jansen and Jurriaan Hage. *How do professionals perceive legacy systems and software modernization?* In Proceedings of the 36th International Conference on Software Engineering, pages 36–47, 2014.

[Kienle 2010] Holger M. Kienle and Hausi A. Müller. *The Tools Perspective on Software Reverse Engineering: Requirements, Construction, and Evaluation*. In Advanced in Computers, volume 79, pages 189–290. Elsevier, 2010.

[Kontogiannis 1998] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller and J. Mylopoulos. *Code Migration through Transformations: An Experience Report*. In Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '98, page 13. IBM Press, 1998.

[Kuhn 2006] Adrian Kuhn and Orla Greevy. *Summarizing Traces as Signals in Time*. In Proceedings IEEE Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006), pages 01–06, Los Alamitos CA, October 2006. IEEE Computer Society Press.

[Kusel 2013] Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Philip Langer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger and Manuel Wimmer. *A survey on incremental model transformation approaches*. In ME 2013–Models and Evolution Workshop Proceedings, volume 1090, pages 4–13. CEUR Workshop Proceedings, 2013.

[Lakhotia 1993] A. Lakhotia. *Rule-based approach to computing module cohesion*. In Proceedings 15th ICSE, pages 35–44, 1993.

[Langelier 2005] Guillaume Langelier, Houari A. Sahraoui and Pierre Poulin. *Visualization-based analysis of quality for large-scale software systems*. In ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pages 214–223, New York, NY, USA, 2005. ACM.

[Lanza 2001] Michele Lanza. *The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques*. In Proceedings of the International Workshop on Principles of Software Evolution, IWPSE'01, pages 37–42, 2001.

[Lanza 2003] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Bern, May 2003.

[Lattner 2004] C. Lattner and V. Adve. *LLVM: a compilation framework for lifelong program analysis and transformation*. In International Symposium on Code Generation and Optimization, 2004. CGO 2004., pages 75–86, 2004.

[Lauder 2012] Marius Lauder, Anthony Anjorin, Gergely Varró and Andy Schürr. *Efficient model synchronization with precedence triple graph grammars*. In Graph Transformations: 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings 6, pages 401–415. Springer, 2012.

[Laval 2011] Jannik Laval. *Package Dependencies Analysis and Remediation in Object-Oriented Systems*. PhD thesis, Université de Lille, 2011.

[Lee 1995] Y. S. Lee and B. S. Liang. *Measuring the coupling and cohesion of an object-oriented program based on information flow*. In In Proceedings of the International Conference on Software Quality (ICSQ), pages 47–57, 1995.

[Lehman 1985] Manny Lehman and Les Belady. Program evolution: Processes of software change. London Academic Press, London, 1985.

[Leroy 2003] Xavier Leroy. *Computer Security from a Programming Language and Static Analysis Perspective*. In P. Degano, editeur, Programming Languages and Systems: 12th European Symposium on Programming, ESOP 2003, volume 2618 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2003.

[Ludewig 2003] Jochen Ludewig. *Models in software engineering–an introduction*. Software and Systems Modeling, vol. 2, no. 1, pages 5–14, 2003.

[Lungu 2006] Mircea Lungu, Michele Lanza and Tudor Gîrba. *Package Patterns for Visual Architecture Recovery*. In European Conference on Software Maintenance and Reengineering (CSMR), pages 185–196, Los Alamitos CA, 2006. IEEE Computer Society Press.

[Malabarba 1999] Scott Malabarba, Premkumar Devanbu and Aaron Stearns. *MoHCA-Java: a tool for C++ to Java conversion support*. In Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002), pages 650–653. IEEE, 1999.

[Martin 2002a] Johannes Martin and Hausi A Muller. *C to java migration experiences*. In Proceedings of the Sixth European Conference on Software Maintenance and Reengineering, pages 143–153. IEEE, 2002.

[Martin 2002b] Johannes Martin and Hausi A Muller. *C to java migration experiences*. In Proceedings of the Sixth European Conference on Software Maintenance and Reengineering, pages 143–153. IEEE, 2002.

[McCabe 1976] Thomas J. McCabe. *A Measure of Complexity*. IEEE Transactions on Software Engineering, vol. 2, no. 4, pages 308–320, December 1976.

[Mellor 2002] Stephen J. Mellor and Marc J. Balcer. Executable uml: A foundation for model-driven architecture. Addison-Wesley Professional, May 2002.

[Merrill 2003] Jason Merrill. *Generic and gimple: A new tree representation for entire functions*. In Proceedings of the 2003 GCC Developers Summit, pages 171–179. Citeseer, 2003.

[Moore 1994] Moore, Rugaber and Seaver. *Knowledge-based user interface migration*. In Proceedings 1994 International Conference on Software Maintenance, pages 72–79. IEEE Comput. Soc. Press, 1994.

[Moore 1996] Melody M Moore. *Rule-based detection for reverse engineering user interfaces*. In Proceedings of WCRE'96: 4rd Working Conference on Reverse Engineering, pages 42–48. IEEE, 1996.

[Mossienko 2003] Maxim Mossienko. *Automated COBOL to Java recycling*. In Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings., pages 40–50. IEEE, 2003.

[Moynihan 1991] Vincent D Moynihan and Peter JL Wallis. *The design and implementation of a high-level language converter*. Software: Practice and Experience, vol. 21, no. 4, pages 391–400, 1991.

[Müller 1997] Hausi A. Müller. *Reverse Engineering Strategies for Software Migration (Tutorial)*. In Proceedings of the 19th International Conference on Software Engineering, ICSE '97, page 659 to 660, New York, NY, USA, 1997. Association for Computing Machinery.

[Murphy-Hill 2009] Emerson Murphy-Hill, Chris Parnin and Andrew P. Black. *How We Refactor, and How We Know It*. In International Conference on Software Engineering (ICSE), pages 287–297, 2009.

[Nagappan 2005] Nachiappan Nagappan and Thomas Ball. *Static Analysis Tools as Early Indicators of Pre-release Defect Density*. In International Conference on Software Engineering, pages 580–586, 2005.

[Namdeo 2022] Basant Namdeo and Ugrasen Suman. *Cost Model for Database Reengineering from RDBMS to NoSQL*. In 2021 4th International Conference on Recent Trends in Computer Science and Technology (ICRTCST), pages 164–168, 2022.

[Newcomb 2005] P. Newcomb. *Architecture-Driven Modernization (ADM)*. In 12th Working Conference on Reverse Engineering (WCRE'05), pages 237–237, 2005.

[Object Management Group 2011] Object Management Group. *Abstract Syntax Tree Metamodel (ASTM) Version 1.0*. Rapport technique, Object Management Group, 2011.

[OMG 2011] OMG. *Knowledge Discovery Meta-Model (KDM)*, 2011.

[Opdyke 1992] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[Overgaard 1980] Mark Overgaard. *UCSD Pascal: a portable software environment for small computers*. In Proceedings of the May 19-22, 1980, national computer conference, pages 747–754, 1980.

[Partsh 1983] H Partsh and R Steinbruggen. *Program transformation systems, ACM Cornput*, 1983.

[Pérez-Castillo 2011] Ricardo Pérez-Castillo, Ignacio Garcia-Rodriguez De Guzman and Mario Piattini. *Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems*. Computer Standards & Interfaces, vol. 33, no. 6, pages 519–532, 2011.

[Pérez-Castillo 2014] R. Pérez-Castillo and M. Piattini. *Analyzing the Harmful Effect of God Class Refactoring on Power Consumption*. IEEE Software, vol. 31, no. 3, pages 48–54, May 2014.

[Plaisted 2013] David A Plaisted. *Source-to-source translation and software engineering*. Journal of Software Engineering and Applications, 2013.

[Plaisted 2021] David A Plaisted and Lee Barnett. *A term-rewriting semantics for imperative style programming: Summary*. In 9th International Symposium on Symbolic Computation in Software Science (SCSS 2021), short and work-in-progress papers, page 40, 2021.

[Porres 2003] Ivan Porres, Turku Centre and Computer Science. *Model Refactorings as Rule-Based Update Transformations*. In 9th International Conference on the Unified Modeling Language, pages 159–174, 2003.

[Rakić 2015] Gordana Rakić. *Extendable and adaptable framework for input language independent static analysis*. PhD thesis, University of Novi Sad (Serbia), 2015.

[Ráth 2008] István Ráth, Gábor Bergmann, András Ökrös and Dániel Varró. *Live model transformations driven by incremental pattern matching*. In Theory and Practice of Model Transformations: First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008 Proceedings 1, pages 107–121. Springer, 2008.

[Razavi 2012] Ali Razavi and Kostas Kontogiannis. *Partial evaluation of model transformations*. In 2012 34th International Conference on Software Engineering (ICSE), pages 562–572. IEEE, 2012.

[Razavian 2012] Maryam Razavian and Patricia Lago. *A Lean and Mean Strategy for Migration to Services*. In Proceedings of the WICSA/ECSA 2012 Companion Volume, WICSA/ECSA '12, page 61 to 68, New York, NY, USA, 2012. Association for Computing Machinery.

[Richner 1999] Tamar Richner and Stéphane Ducasse. *Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information*. In Hongji Yang and Lee White, editeurs, Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99), pages 13–22, Los Alamitos CA, September 1999. IEEE Computer Society Press.

[Sakamoto 2013] Kazunori Sakamoto. *A framework for analyzing and transforming source code supporting multiple programming languages*. In Proceedings of the 12th annual international conference companion on Aspect-oriented software development, pages 35–36, 2013.

[Sánchez Ramán 2010] Óscar Sánchez Ramán, Jesús Sánchez Cuadrado and Jesús García Molina. *Model-driven Reverse Engineering of Legacy Graphical User Interfaces*. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, pages 147–150. ACM, 2010.

[Selmadji 2020] Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Rahina Oumarou Mahamane, Pascal Zaragoza and Christophe Dony. *From monolithic architecture style to microservice one based on a semi-automatic approach*. In 2020 IEEE International Conference on Software Architecture (ICSA), pages 157–168. IEEE, 2020.

[Sendall 2003] Shane Sendall and Wojtek Kozaczynski. *Model transformation: The heart and soul of model-driven software development*. IEEE software, vol. 20, no. 5, pages 42–45, 2003.

[Shah 2011] Eeshan Shah and Eli Tilevich. *Reverse-engineering user interfaces to facilitate porting to and across mobile devices and platforms*. In Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, \& VMIL'11, pages 255–260. ACM, 2011.

[Siikarla 2008] Mika P. Siikarla and Tarja J. Systa. *Decision Reuse in an Interactive Model Transformation*. In 2008 12th European Conference on Software Maintenance and Reengineering, pages 123–132, 2008.

[Silva 2013] Carlos E. Silva and José C. Campos. *Combining static and dynamic analysis for the reverse engineering of web applications*. In Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, page 107. ACM Press, 2013.

[Sneed 1999] Harry M. Sneed. *Risks Involved in Reengineering Projects*. In Proceedings of the 6th Working Conference on Reverse Engineering (WCRE). IEEE, 1999.

[Sneed 2004] H.M. Sneed. *A cost model for software maintenance & evolution*. In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, pages 264–273, sep 2004.

[Sneed 2011] H Sneed. *Migrating from COBOL to Java–A Report from the Field*. IEEE Proc. of 26th ICSM, Computer Society Press, Temesvar, Ro, page 122, 2011.

[Sneed 2020] Harry M Sneed and Chris Verhoef. *Cost-driven software migration: An experience report*. Journal of Software: Evolution and Process, page e2236, 2020.

[Software 1997] Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam. Unified modeling language (version 1.1). Rational Software Corporation, September 1997.

[Stroulia 2002] E. Stroulia and T. Systä. *Dynamic analysis for reverse engineering and program understanding*. SIGAPP. Applied Computing Review, vol. 10, no. 1, pages 8–17, 2002.

[Su 2009] Xing Su, Xiaohu Yang, Juefeng Li and Di Wu. *Parallel iterative reengineering model of legacy systems*. In 2009 IEEE International Conference on Systems, Man and Cybernetics, pages 4054–4058. IEEE, 2009.

[Subramanian 1996] Girish H Subramanian and George E Zarnich. *An examination of some software development effort and productivity determinants in ICASE tool projects*. Journal of Management Information Systems, vol. 12, no. 4, pages 143–160, 1996.

[Taivalsaari 1995] Antero Taivalsaari, Roland Trauter and Eduardo Casais. *Workshop on Object-Oriented Legacy Systems and Software Evolution*. SIGPLAN OOPS Mess., vol. 6, no. 4, page 180 to 185, 1995.

[Terekhov 2000] A. A. Terekhov and C. Verhoef. *The realities of language conversions*. IEEE Software, vol. 17, no. 6, pages 111–124, November 2000.

[Trudel 2012] Marco Trudel, Carlo A Furia, Martin Nordio, Bertrand Meyer and Manuel Oriol. *C to OO translation: Beyond the easy stuff*. In 2012 19th Working Conference on Reverse Engineering, pages 19–28. IEEE, 2012.

[Tufte 1997] Edward R. Tufte. Visual explanations: images and quantities, evidence and narrative. Graphics Press, Cheshire, CT, USA, 1997.

[Tufte 2001] Edward R. Tufte. The visual display of quantitative information. Graphics Press, 2nd édition, 2001.

[Ulrich 2010] William M Ulrich and Philip Newcomb. Information systems transformation: architecture-driven modernization case studies. Morgan Kaufmann, 2010.

[Verhaeghe 2019] Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Abderrahmane Seriai, Laurent Deruelle, Stéphane Ducasse and Mustapha Derras.

GUI Migration using MDE from GWT to Angular 6: An Industrial Case. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19), pages 579–583, Hangzhou, China, 2019.

[Verhaeghe 2021a] Benoît Verhaeghe. *Incremental Approach for Application GUI Migration using Metamodel*. PhD thesis, Université de Lille, 2021.

[Verhaeghe 2021b] Benoît Verhaeghe, Anas Shatnawi, Abderrahmane Seriai, Anne Etien, Nicolas Anquetil, Mustapha Derras and Stephane Ducasse. *From GWT to Angular: An Experiment Report on Migrating a Legacy Web Application*. IEEE Software, 2021.

[Verhaeghe 2022] Benoît Verhaeghe, Anas Shatnawi, Abderrahmane Seriai, Anne Etien, Nicolas Anquetil, Mustapha Derras and Stéphane Ducasse. *A Hybrid Architecture for the Incremental Migration of a Web Front-end*. In Proceedings of the 17th International Conference on Software Technologies - ICSOFT, pages 101–110. INSTICC, SciTePress, 2022.

[Visser 2004] Eelco Visser. *Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9*. In C. Lengauer*et al.*, editeurs, Domain-Specific Program Generation, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[Visser 2005] Eelco Visser. *Transformations for Abstractions*. Rapport technique UU-CS-2005-034, Department of Information and Computing Sciences, Utrecht University, 2005.

[W3C 1999] W3C. *XSL Transformations (XSLT) Version 1.0*, nov 1999.

[Ware 2000] Colin Ware. Information visualization: perception for design. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[Wegner 1987] Peter Wegner. *Dimensions of Object-Based Language Design*. In Proceedings of OOPSLA '87, volume 22, pages 168–182, December 1987.

[Wettel 2007] Richard Wettel and Michele Lanza. *Visualizing Software Systems as Cities*. In Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis), pages 92–99, 2007.

[Wettel 2008] Richard Wettel and Michele Lanza. *Codecity: 3d visualization of large-scale software*. In Companion of the 30th international conference on Software engineering, pages 921–922, 2008.

[Williams 2012] James R. Williams, Richard F. Paige and Fiona A. C. Polack. *Searching for Model Migration Strategies*. In Proceedings of the 6th International Workshop on Models and Evolution, ME ’12, page 39 to 44, New York, NY, USA, 2012. Association for Computing Machinery.

[Wirth 1966] Niklaus Wirth and Charles Antony Richard Hoare. *A contribution to the development of ALGOL*. Communications of the ACM, vol. 9, no. 6, pages 413–432, 1966.

[Wu 1997] Bing Wu, Deirdre Lawless, Jesus Bisbal, Ray Richardson, Jane Grimson, Vincent Wade and Donie O’Sullivan. *The butterfly methodology: A gateway-free approach for migrating legacy information systems*. In Proceedings. Third IEEE International Conference on Engineering of Complex Computer Systems (Cat. No. 97TB100168), pages 200–205. IEEE, 1997.

[Wysseier 2005] Christoph Wysseier. Interactive 3-D visualization of feature-traces. Master’s thesis, University of Bern, Switzerland, November 2005.

[Yasumatsu 1995] Kazuki Yasumatsu and Norihisa Doi. *SPiCE: A System for Translating Smalltalk Programs Into a C Environment*. IEEE Trans. Software Eng., vol. 21, no. 11, pages 902–912, 1995.

[Zaragoza 2022] Pascal Zaragoza. *Model-Driven Software Migration of Legacy Applications to Micro-Services Architectures*. PhD thesis, Informatique Montpellier, 2022.

[Zhang 2004] Zhuopeng Zhang and Hongji Yang. *Incubating services in legacy systems for architectural migration*. In 11th Asia-Pacific Software Engineering Conference, page 196 to 203. IEEE, 2004.

[Zhou 2007] Hong Zhou, Jian Kang, Feng Chen and Hongji Yang. *OPTIMA: an ontology-based PlaTform-specIfic software migration approach*. In Seventh International Conference on Quality Software (QSIC 2007), pages 143–152. IEEE, 2007.

[Zou 2001] Ying Zou and Kostas Kontogiannis. *A framework for migrating procedural code to object-oriented platforms*. In Proceedings Eighth Asia-Pacific Software Engineering Conference, page 390 to 399. IEEE, 2001.

[Zou 2005] Ying Zou. *Quality driven software migration of procedural code to object-oriented design*. In 21st IEEE International Conference on Software Maintenance (ICSM’05), pages 709–713. IEEE, 2005.